Parallel persistent object-oriented simulation with applications

Burdoff, Christopher

Award date:
1993

Awarding institution:
University of Bath

Parallel Persistent Object-Oriented Simulation With Applications

submitted by

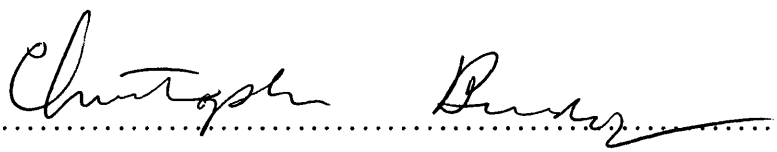
Christopher Burdorf

for the degree of Ph.D

of the

University of Bath

1993

Signature of Author 

Christopher Burdorf

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Summary

Computer simulation is a diverse field covering a wide range of models and applications encompassing many disciplines. As the complexity of simulation models have grown, there has been an increased demand for programming environments which help the application developer organize and modify his or her system while providing increased performance.

Highly complex simulations utilizing tens of thousands of objects need support for the analysis of these objects and their behaviour. *Persistence* supports this goal by providing an underlying layer which manages objects and their simulation histories *transparently* between primary memory and secondary memory. These objects can then be analysed by the developer once the simulation has completed. On successive executions of the application, the persistent objects are then automatically loaded from secondary memory to primary memory by the persistent object system on a demand-driven basis without requiring the application programmer to implement any file-handling code.

The demand for improved *performance* is important with some modern simulations requiring weeks or even months of computing time to execute. While processors continue to get more powerful every year, there is alway a need to reduce the execution times of large simulations. Thus *parallelism* is merged with persistence to meet performance needs. First sequential algorithms for simulation and persistent object management are improved to provide efficient sequential codes. Then, as this thesis shows, these sequential codes can then be parallelized resulting in improved performance.

Since simulation models are so diverse, it is not possible to support all of them in one system. Nevertheless, some of the most widely used simulation models are supported by the Persistent Simulation Environment (PSE) as described in this thesis. They include discrete-event, process-based, connectionist, and Petri net models. The use of these models and the applications developed under them illustrate that the merging of persistence and parallelism has advantages across several simulation models by providing both improved performance and a seamless interface between the program and secondary memory.

Contents

List of Figures

Acknowledgements

Many thanks to everyone who helped me. Thanks to SERC for three years of support and for providing a great amount of academic freedom. Thanks to Pete Broadberry, Keith Playford, and Simon Merrall for EuLisp support. Also, much thanks to Professor Fitch for constructive criticism and encouragement. Finally, thanks to Kay Marie Sutcliffe for being there.

Chapter 1

Introduction

Object-Oriented simulation began with Simula [Dah66] and has been extended in languages like ROSS [McA82], ModSim [Her92a] and PSE [Cam91]. PSE (the Persistent Simulation Environment) was the first simulation language that supported persistent objects [Atk89]. Persistence extends object-oriented programming such that objects reside in both primary and secondary memory. Persistent object systems provide a seamless integration between a database and a programming language through management of object creation, reference, deletion, and modification. PSE was originally designed to augment a contemporary object-oriented language with discrete-event and process-based simulation facilities equaling those found in Simscript[Rus79] and Simula, and to couple an object-oriented simulation language tightly with a secondary storage facility to achieve the persistence of simulation objects. This thesis describes the merging of persistent object-oriented simulation with parallelism and applies it to several simulation paradigms to support more than one type of application and hybrid applications. Large-scale simulations modelling real-world behaviour often require the use of more than one paradigm. Therefore, it is practical for a simulation programming environment to provide support for several models and allow them to be incorporated. The paradigms supported by PSE, upon the completion of this thesis, include discrete-event, process-based, connectionist, and Petri nets. These models can be combined by loading the separate libraries into a single image.

This thesis shows that general concepts like persistent objects and parallelism can be merged to support different simulation models through a set of utilities supplied to the simulation programmer. The advantage of having persistent objects in a simulation environment is that it supports the storage and reuse of simulation objects in a transparent

manner which allows perusal of simulation behaviour after program execution. Parallelism is used to improve the performance of simulations.

1.1 Background and Definitions

This section provides background and definitions on object-oriented simulation, persistent object systems, and parallel discrete-event simulation which will be referred to later on in this thesis.

Three concepts characterize *object-oriented* languages:

1. Encapsulation: behaviours or generic functions form the only possible interface to communicate with objects.

2. Polymorphism: a behaviour or generic function can be defined to do different operations for each class or set of classes it is defined for.

3. Inheritance: a class can inherit attributes and operations from another class.

Persistent object systems (POS) [Atk89], as stated previously, provide a seamless integration between a programming language and a database. Persistent objects reside in secondary memory as well as primary memory and modifications made to them in primary memory are propagated to secondary memory by the persistent object system in a manner that is *transparent* to the application programmer.

Some of the advantages of persistent systems listed by Morrison and Atkinson [Mor90] include:

1. reduced complexity

2. reduced code size and time to execute

3. data outlives the program

Firstly, complexity is reduced for the application builders, because with persistent systems, there is no distraction for the programmer in dealing with the complexity of managing the database. He or she need only consider the complexities involved in the mapping between the programming language and the problem to be solved. Secondly, persistent systems reduce code size, because the application program need not contain code concerned with the explicit movement of data between primary and secondary memory. Also, the time

to execute is reduced, because *only objects required* by the system get loaded into primary memory. Finally, the data outlives the program, because it resides in a database. According to Atkinson and Morrison these are the reasons why persistence is advantageous. It is due to the advantages provided by persistence that it was chosen to be one of the main focuses of this thesis.

Object-oriented simulation [Bou92] consists of first forming classes of objects with common attributes and behaviours. Classes can refer to real object categories (workstation, waiting queue) or abstract object categories (operation, routing). The modelling governs the programming of the system, so it must be accurate, though it is usually updated during development using stepwise refinement. Object-oriented simulation assumes the representation of system knowledge: objects' characteristics, their behaviours, and interactions between them. In object-oriented simulation, methods or generic functions are used to program state transition logics. The state transition logic is the plan that the simulation is to follow throughout its execution. The state transition logic can be realized as events or processes which determine which objects' states get affected. At each processing of an event or process, when a corresponding state transition logic is launched, associated events or processes are activated through associated generic function calls on them. Time stamps are associated with event and process invocation to allow the simulation programmer to specify the ordering of events over time. For example, the behaviour *change-direction* on class *airplane* needs to have a time associated with it, so the simulation can know when in time the airplane is to alter its direction.

Parallel discrete-event simulation has two main approaches: conservative and optimistic. Time Warp [Jef85a, Jef85b] is the most commonly used optimistic algorithm. The basic device used by Time Warp is to impose a *virtual* time order for every process. This ordering is achieved by having each process queue its incoming messages by time-stamp order rather than arrival order. In this way, a process can be thought of as working along its input queue, increasing its local virtual time (LVT) (the time-stamp of the message currently being handled by a process) to the time-stamp of each message as it gets to it. When a message arrives whose time-stamp is smaller than the process's LVT, the message lands in that part of the queue already processed, thereby causing a rollback.

Time Warp solves the problem of ensuring the correct order of messages by *unsending* them. Each real message has a corresponding antimessage which, when sent to the same destination as the original positive message, serves to annul it. It is crucial that the antimes-

sage carry the same time-stamp as its positive corresponding message, because the arrival of the antimessage must invalidate any work performed by the recipient from that time on, and cause the recipient to rollback to that point if necessary. Should the recipient roll back in response to the antimessage, it will send more antimessages to yet other processes. In this way, the rollback will propagate to the other affected parts of the system as desired. I worked on the development of a Time Warp system while at RAND corporation. My work involved the development of scheduling algorithms [Bur90] and static and dynamic load balancing [Bur93a].

Conservative techniques for *parallel discrete-event* simulation require that the events for each object execute in time-stamp order. This requirement restricts communicating objects from overtaking one another. Therefore, unlike optimistic methods, there is no rollback.

Conservative techniques can be summarized as follows [Cha81, Fuj90]: if a process contains an unprocessed event E1 with time-stamp T1 and that process can determine that it is impossible for it to receive later another event with time-stamp smaller than T1, then the process can safely proceed with E1 because it can guarantee that doing so will not later result in a violation of the local causality constraint. Processes containing no safe events must block; this can lead to deadlock situations if appropriate precautions are not taken. The sequence of messages on an input queue must be in nondecreasing order. Each queue has a clock associated with it that represents the time of either the first message on the queue or if there are no messages, the time of the last message processed. Each object repeatedly selects the queue with the smallest time-stamp and processes the message on it if there is one. Otherwise it blocks and waits for a message to arrive on that queue. The protocol guarantees that each process will only process events in time-stamp order.

The merging of persistent object-oriented simulation with parallel discrete-event simulation in support of several models and support for the hybridization of those models is the goal of this thesis. Large-scale models often require the merging of several models, and parallelism and persistent objects provide benefits, as described previously, in a simulation programming environment.

1.2 Related Work

Persistent systems date back to the early 1980s [Atk83]. There is now a large number of commercial and research programming languages that support persistence. A model for

persistent objects was described by Rowe [Row86] which formed the basis for the Picasso system and is the model that PSE's persistent object system (the one used in this thesis) is based on.

The Time Warp operating system [Jef87] and the RAND Time Warp system [Bur89] both provided programming environments that supported parallel discrete-event simulation using the Time Warp method of optimistic concurrency control. It should be noted that these systems only support event-based simulation models.

Event and process-based simulation, provide fairly limited support on their own for complex modelling. As a result, there has been a trend towards the use of higher level representations and knowledge-bases in conjunction with discrete-event simulation to provide the kind of powerful tools needed to support complex models.

In the simulation world, there has been a trend towards the support of various models and the hybridization of those models. Large complex simulations often require expert knowledge at decisive points during execution. As a result, there have been a large number of systems recently that incorporate several models to support complex simulations. Intelligent agent systems use distributed knowledge bases to do planning for simulation activities. Norrie and Kwok apply an intelligent agent system to do the planning for a simulation of an automated guided vehicle [Nor92]. Likewise, Balducelli and Vicoli [Bal92] have used a knowledge base interface to predict faults in the simulation of an electrolytic cell.

Ali [Ali93] points out the large amount of computation required in symbolic processing-based expert systems. Ali argues in favor of using connectionist-based expert systems to interface with simulations, because they require far less computation to solve similar tasks. For this reason, a connectionist representation language was added to PSE as a part of the work for this thesis. The connectionist representation also has explicitly parallel properties which make it a good model to map onto a parallel machine which will also be shown in this thesis.

Petri nets were chosen to be added to PSE for this thesis, because they can be used to produce numerical estimations on the performance of parallel systems. Petri nets are a graphical and mathematical modelling tool applicable to many systems [Mur89]. They are a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic. As a graphical tool, Petri nets can be used as a visual communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets

to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behaviour of systems. Petri nets can be used by both practitioners and theoreticians.

Petri nets are a higher-level representation which are currently widely used for simulations involving performance evaluation. Marsan [Mar87, Mar86, Mar89] has used stochastic Petri nets extensively for the performance evaluation of parallel computing systems. Likewise, Petri-net simulations are being combined with knowledge-based systems to support transitions that require expert knowledge. A hybrid Petri-net and knowledge-based system has been used to model the performance of on-line scheduling in a flexible manufacturing system (FMS) [Hat91]. The developers gave the following reasons for choosing Petri nets:

> We can explicitly describe the causal relation of uncertain events by using places, transitions, and arcs. Therefore, using stochastic Petri nets, we can construct the model of an FMS more easily than using the other models.

The FMS Petri net simulation also uses an on-line rule-based scheduling system which selects a transition to be fired from a set of conflicting ones. Similarly, Lee [Lee91] has described a model which incorporates Fuzzy rule sets and Petri nets. Also, Abellard et al [Abe93] have developed a Petri net simulation that relies on a *neural network interface* to make scheduling decisions. Since Petri nets are widely used and are combined with knowledge bases to form hybrid simulations, they have been chosen to be supported by PSE for this thesis.

In response to the increased use of higher-level representations, knowledge bases, and hybrids, there has been recent development of tools that support these kinds of models. Simpack and Simpack++ are simulation tools developed at the University of Florida. They both support continuous and discrete-event simulation as well as *Petri* nets, Markov models, Network simulation, and Finite State Automata simulations. Simpack++ is an object-oriented version of Simpack. Simpack's Petri net package supports only simple Petri nets which are not as widely used as stochastic or Colored Petri net models which are supported under PSE as a result of the work done for this thesis. Likewise, Soubra et. al. [Sou93] describe the design of an intelligent simulation environment which supports the sharing of common data and the incorporation of expert systems and diagnosis tools with utilities for various simulation models. While it is not proposed by Soubra, persistence is a means of

sharing common data between models, because the data can be readily accessed from a database by different simulations. Also, Laret [Lar93] reviews some of the previous and ongoing projects involving the development of simulation tools that support model libraries which provide support for a vast range of simulation modelling. He describes several systems which support qualitative, numerical, and experimental models as well as provide interfaces for knowledge-base consultation. *Laret argues that large-scale simulations require the integration of several models and knowledge bases.*

Modsim [Her92a] is a simulation language which supports discrete-event and process-based simulation models. It also has an interface to Prolog called Modlog [Whi92b] which can be used to develop a knowledge base for consultation by a simulation. Modsim also has a facility for so-called persistent objects [Her92b], but inspection of its usage reveals that it really only allows for the inclusion of object-oriented database code. Modsim's support for persistence is seriously lacking in a seamless interface in that persistent objects must be treated differently from non-persistent objects at all the stages of their usage. Modsim was designed to be executed under the Time Warp operating system, and some Modsim applications have been ported and execute under Time Warp. However, according to Herring, Modsim programs need to be significantly altered to execute efficiently under the Time Warp operating system.

PSE, as developed for this thesis, has several advantages over Modsim. Firstly, PSE's persistent object system provides a much more seamless interface between secondary and primary memory in that modifications get propagated to the database by the underlying persistent object system in a manner which is transparent to the applications programmer. For example, in PSE when a slot in a persistent object is modified by the program, PSE's underlying persistent object system propagates the modification to the database without requiring any instruction from the programmer. Also in PSE, when a programmer makes a reference to to a persistent class or object that resides in the database but not in the virual image, the persistent object system loads and instantiates it without requiring any instructions from the programmer. However, in Modsim, the programmer must keep track of which objects and classes are in the database and not in the virtual image, because ModSim requires that one explictly load objects from the database before referencing them. Likewise in Modsim, the programmer write code which handles the propagation of object modifications to the database. Also, PSE now has a connectionist representation for the development of knowledge bases which can be accessed from a simulation. The advantage

of using a connectionist knowledge base over one developed using Modlog is that the connectionist model requires less computation and is less difficult to port to a parallel machine. Due to their explicitly parallel nature, connectionist models are easier to port parallel machines than are symbolic pattern-directed inferencing mechanisms such as OPS-5 [Bro85]. Also, since connectionist systems do inferencing based on arithmetic operations, it seems likely that they have better performance than symbolic pattern-directed systems, because computers perform much better doing numerical calculations than they do for when performing symbolic pattern matching. PSE has support for the development of Petri-net models, but ModSim lacks any support for a higher-level representation like Petri nets. Finally, PSE supports parallel execution of Petri nets, connectionist knowledge bases, and process-based simulation. ModSim has no support for parallel Prolog. However, some Modsim simulations have been executed on the Time Warp operating system [Jef87] which supports parallel simulation, but it is not a part of the language itself.

1.3 PSE

I have been involved with the design and implementation of PSE from the beginning. The first two years of its development was undertaken when I was at RAND corporation and was done under the direction of Stephanie Cammarata and was written in Allegro Common Lisp. The system developed at RAND consisted of the persistent object kernel, discrete-event, and process-based simulation utilities. Upon my arrival at Bath, I ported the code to EuLisp [Pad91] to take advantage of its support for parallel programming. The port was nontrivial and is described in Chapter 2.

Once the port, which took 4 months, was completed, I progressed onto merging it with parallelism to improve performance and extended it to support various models to broaden the types of applications it could support, to be consistent with developments in the field (as described in section 1.2), demonstrate its effectiveness, and most importantly, show that persistence and parallelism can be used in support of various simulation paradigms. As shown in section 1.2, simulations are written in a variety of paradigms which are often combined to form hybrid models. Thus, a programming environment for simulation should be extended to support a wide range of modelling capabilities. Secondly, simulations tend to be large and require an immense amount of computation; therefore for the environment to be usable it must be optimized for performance by producing efficient sequential codes

and through parallelization as is shown in this thesis. Finally, to show the effectiveness of the system, it should be demonstrated that it can be used to develop large-scale models, produce results, and execute in a reasonable amount of time. In this thesis, large-scale simulations have been implemented and tested using PSE to show that these goals have been attained.

1.4 Thesis Organization

The specific activities undertaken to achieve these goals were numerous. Chapter 2 describes PSE's persistent object system in detail and explains the difficulties encountered in the port from Common Lisp to EuLisp. EuLisp's thread facility provides a basic primitive for parallel programming that can be used for the development of higher level parallel programming constructs to be used by an application. The motivation being that parallel programming could improve PSE's *performance*. Chapter 3 describes a set of algorithms for replacement of objects in the persistent object cache that were developed, and experiments were conducted to determine which one would best improve its performance, because it makes no sense to apply parallelism to inefficient sequential code. The algorithm which produced the best results was then incorporated into PSE. Chapter 4 describes the discrete-event and process-based simulation constructs that were ported from the Common Lisp version of PSE to EuLisp by the author while at Bath University.

Chapter 5 describes an extension that was then made to PSE to handle concurrent process-based simulation under conservative concurrency control. An example is presented which was implemented in the system for the cloning of objects under conservative protocols. One of the criticisms made for conservative mechanisms is that by definition it does not allow dynamic object creation. The cloning facility presented in chapter 5 is a solution to that problem and makes the conservative mechanism more flexible and extends its modelling capabilities to handle models like networking simulations where servers get added dynamically.

Chapters 6 and 7 describe another extension that was made by the author at Bath to support a connectionist representation language called POCONS (Persistent Object-based CONnectionist Simulator) which can be used on its own or can be interfaced with a simulation for consultation purposes as described in section 1.2. Connectionist models provide a mechanism for representing knowledge through connections between neurons. Those con-

nections are weighted to represent the certainty factors between semantic relationships. The extension to handle connectionist systems was made to show that persistent objects and parallelism can be applied to support the widespread use of connectionist systems in the development of knowledge bases used for consultation by a simulation.

To improve its knowledge-representation abilities, POCONS was then further extended by the author to have a construct which supported implication to support rule-based programming. A circuit analysis program was implemented in POCONS and served as an application test bed for the system. Parallel techniques on SIMD and MIMD machines were implemented and improved its *performance* on the circuit analysis program. POCONS was then further extended to include a utility for storage and reuse of connectionist objects which improved the *performance* of the system, because information developed during the initialization of the system could be stored in persistent objects and when reloaded, it eliminates the need for retraining. Then, a mechanism was introduced into POCONS which further extended it to support chaotic reasoning, because in certain systems (like financial markets) a user may want the model to exhibit chaotic behaviour. This facility for chaotic neural networks was then used in a simulation of commodity price fluctuations. Results from the simulation showed similar behaviour to that of the precious metals' market it was modelling. These extensions to POCONS improved its capabilities for the development of knowledge-based systems which can be interfaced with a simulation. Also, the use of parallelism improved its performance so that it could be used for large-scale simulations without requiring an overwhelming amount of computing time as is the case for many symbolic-based knowledge representation systems.

Chapter 8 describes another module that was then added to the EuLisp version of PSE to include a representation language for the description of Petri nets called Per-Trans. The Per-Trans library was added to PSE to support the kinds of models described in section 1.2 and to make PSE a system that could handle the wide range of models which are currently used by simulation developers.

A large-scale ten page distributed virtual shared memory system was developed and simulated using Per-trans which proved to be an exhaustive test. Results of the simulation as well as others are presented in this thesis. Techniques were then developed and tested which improved the performance of the object-oriented simulation produced by Per-trans. These techniques can be applied to object-oriented simulation in general. Also, utilities for parallel simulation of Petri nets were implemented and their utilization improved the

performance of Per-trans simulations.

Thus, there are various paradigms and hybrids that are now supported in PSE, and the application of parallelism and persistent objects, as will be shown in the following chapters, has been effective.

Chapter 2

Persistent Objects in PSE

This chapter covers the persistent object system component of the Persistent Simulation Environment (PSE) and its port from Common Lisp to EuLisp. Later on it will be described how PSE's persistent object system is merged with parallelism in utilities that support various simulation paradigms and hybrids. However, first the background will be laid; first by a description of the persistent object system in this chapter followed by a performance evaluation which determines which caching algorithm to use in the persistent object system as is described in the following chapter. The persistent object system manages the seamless integration between primary and secondary memory by initiating read and write requests as a result of slot accesses and modifications.

The discussion in this chapter will begin with general concepts of object-oriented programming and will lead into a discussion on persistent objects in general. The persistent object system in PSE will then be described in detail, and the chapter will be concluded after discussing problems encountered in porting PSE to EuLisp.

2.1 Objects and Classes

Amongst the concepts of object-oriented languages, the one that is of the most fundamental importance is object classes. In object-oriented languages, object classes are abstract data types that organize both data structures and functions. Each class specifies its relation to existing classes by supplying a reference to the base class, or class from which it is derived. The object class is used in a similar manner as a conventional type allowing instances of its class to be instantiated at the runtime of the program.

The notion of an object class hierarchy dates back to Simula [Dah67]. An object class hierarchy tree grows as new classes are added to the system. Since each new class is a superset of its base class, it inherits selected data and functions, meaning that the derived class, when given access by the inheritance rules, can call any of its base class functions, or modify any of the data.

2.2 Persistent Objects in General

Persistent object systems (POS) [Atk89] provide a seamless integration between a programming language and a database. The POS requires a cache to hold objects that have been loaded into primary memory to avoid the need for reloading an object each time it is accessed. The persistent object cache can be viewed as similar to the working set in a virtual memory system. The size of the cache has to be limited so as not to swamp the runtime system with more objects than can exist without exceeding the size of swap space. Also, since objects may be shared with other users, it is not desirable for any one user to have control over too many objects at a given time, and therefore, caches can also be useful to limit the number of objects owned by a user.

As mentioned in the introduction, some of the advantages of persistent systems as described by Morrison and Atkinson [Mor90] include:

1. reduced complexity

2. reduced code size and time to execute

3. data outlives the program

Firstly, complexity is reduced for the application builders, because with persistent systems, there is no distraction for the programmer in dealing with the complexity of managing the database. He or she need only consider the complexities involved in the mapping between the programming language and the problem to be solved. Secondly, persistent systems reduce code size, because the application program need not contain code concerned with the explicit movement of data between primary and secondary memory. Also, the time to execute is reduced, because *only objects required* by the system get loaded into primary memory. Finally, the data outlives the program, because it resides in a database. According to Atkinson and Morrison, these are the reasons why persistence is advantageous.

The persistent component of PSE does significantly reduce the complexity of the application program, because the underlying persistent object system handles all the database operations involved with the loading and instantiation of objects as well as the automatic propagation of slot modifications to the dataabase. As a result, the simulation programmer can focus solely on the problem to be solved and is insulated from database complexities. The logical result of the elimination of database complexities is the reduction of code size, because all that extra application code used to handle database operations is no longer necessary. However, as to Atkinson and Morrison's claim of time to execute, the time to load objects from a database does considerably slow down a simulation. However, since in PSE they are loaded on demand, if there are a lot of objects, it can significantly reduce the amount of time to execute the simulation, because if only the objects that are needed get loaded it won't swamp the virtual memory system and cause it to thrash. Evidence to support this claim was supported by an experiment I conducted using the Common Lisp version of PSE while at RAND. I had a simulation that used 10,000 objects. Loading all of the objects made the virtual image too large and caused thrashing in the virtual memory system. By using persistence, objects were only loaded on demand, and garbage collections of objects that were no longer needed (to be discussed in the next chapter) kept the virtual image from becoming too large to thrash the virtual memory system.

Morrison and Atkinson also state that the goal of a persistent object system is to support four major functions: sharing, maintaining, inspecting, and reusing of objects. Sharing allows the concurrent use of persistent objects by more than one application program, similar to a database management system which supports access by multiple programs. Object maintenance (insertion, deletion, and updating of simulation objects) can be performed in virtual memory during simulation processing, or through maintenance routines applied directly to objects in the persistent object repository, external to any simulation program. Objects modified during simulation processing will be transparently updated in the persistent repository so that consistency is maintained between virtual objects in the simulation and secondary storage persistent objects. Likewise, objects can be retrieved and inspected during simulation processing and at any time before or after the simulation. Finally, with a persistent object repository, simulation objects can be reused without recreating and initializing objects for each simulation trial. Thus, reusablility can result in improving performance of the system as will be shown in section 7.1.2 of this thesis.

2.3 Persistent Objects and Database Management Systems

Ullman [Ull80] defines a database manangement system as the software that allows one or many persons to use and/or modify data that is more or less stored permanently. Ullman also writes:

> A major role of the DBMS is to allow the user to deal with the data in abstract terms, rather than as the computer stores the data. In this sense, the DBMS acts as an interpreter for a (very) high-level language such as APL, ideally allowing the user to specify what must be done, with little or no attention on the user's part to the detailed algorithms or data representation used by the system. However, in the case of a DBMS, there may be even less relationship between the data as seen by the user and as stored in the computer, than between APL arrays and the representation of these arrays in memory.

Ullman's description so far applies to persistent object systems as well as database systems. However, he goes on to describe other functions which a database system should be expected to carry out like *security*, *integrity*, *synchronization*, and *crash protection and recovery*. These operations are not necessarily required for a persistent object system, but they are supported by Picasso's shared object hierarchy [Row86] and other persistent object systems. Usually, as in Picasso, persistent object systems are interfaced to a database management system, so it can carry out those tasks.

Unlike a database management system, persistent object systems manage a seamless interface between a program and a database. As stated previously in this chapter, the persistent object system simplifies an application programmer's task in that it handles all references to a database automatically. The persistent object system manages an interface between a programming language and a database such that classes, objects, and sometimes methods will be maintained between primary memory and secondary memory on demand and in a manner which is transparent to the application programmer.

2.4 Persistence in PSE

PSE's persistent object system supports sharing, maintaining, and inspecting of objects. Objects can be shared between applications. However, PSE does not support the concurrent sharing of persistent objects, because it involves issues of transaction management which is

not one of the primary goals of this work. Nevertheless, other persistent object languages like Gemstone [But91] and Picasso already support concurrent sharing of objects, because they are interfaced to database management systems which handle transaction synchronization for them. A database management system was not available on the Stardent at the time of development of this system, and there was no reason to reinvent the wheel. Thus, the author's own object management system was used to handle the management of secondary storage, but no support of concurrent sharing of objects was implemented.

In general, an object which is declared to be persistent is retained in secondary storage after program execution terminates. In PSE, once a class has been declared to be persistent, instances of that class will automatically be made persistent. However from the programmer's perspective, persistent objects in PSE are referenced identically to non-persistent simulation objects. Furthermore, fetching and instantiating of a persistent object from secondary storage is performed transparently by the underlying PSE kernel. The kernel implementation of PSE is based on Rowe's [Row86] SOH (shared object hierarchy) methodology.

PSE is composed of the following components pictured in Figure 2-1: persistent object and class files, object space, and object and class directories. The object and class (not pictured) files store an *ascii* representation of the objects and classes in secondary storage. Object space denotes the area in main memory where the object and class structures reside, and the object and class (not pictured) directories contains one *handle* per object. The directories store handles which can be retrieved through an identifier (integer value) which is assigned to them upon creation. The handle is a pseudo-object which is used as a reference point for a persistent object. The handle for an object contains the class wrapper for it, so that as far as the program and programmer are concerned, it is the object. However, upon slot access or modification, the persistent object system will use information stored in the handle concerning the object's location on the disk to load the object from the database or propagate a modification to the database as is necessary.

The object and class handles contain meta-information about the objects and classes and always remain in primary memory. A handle (see figure 2-2) includes information such as a pointer to the object's memory location (which is "nil" if the object is not in the object space), the object's location in the object file, whether or not the object has been modified, and the object's update mode. The update mode indicates how the object will be modified on disk. If the mode is *direct-update* the object will be updated immediately

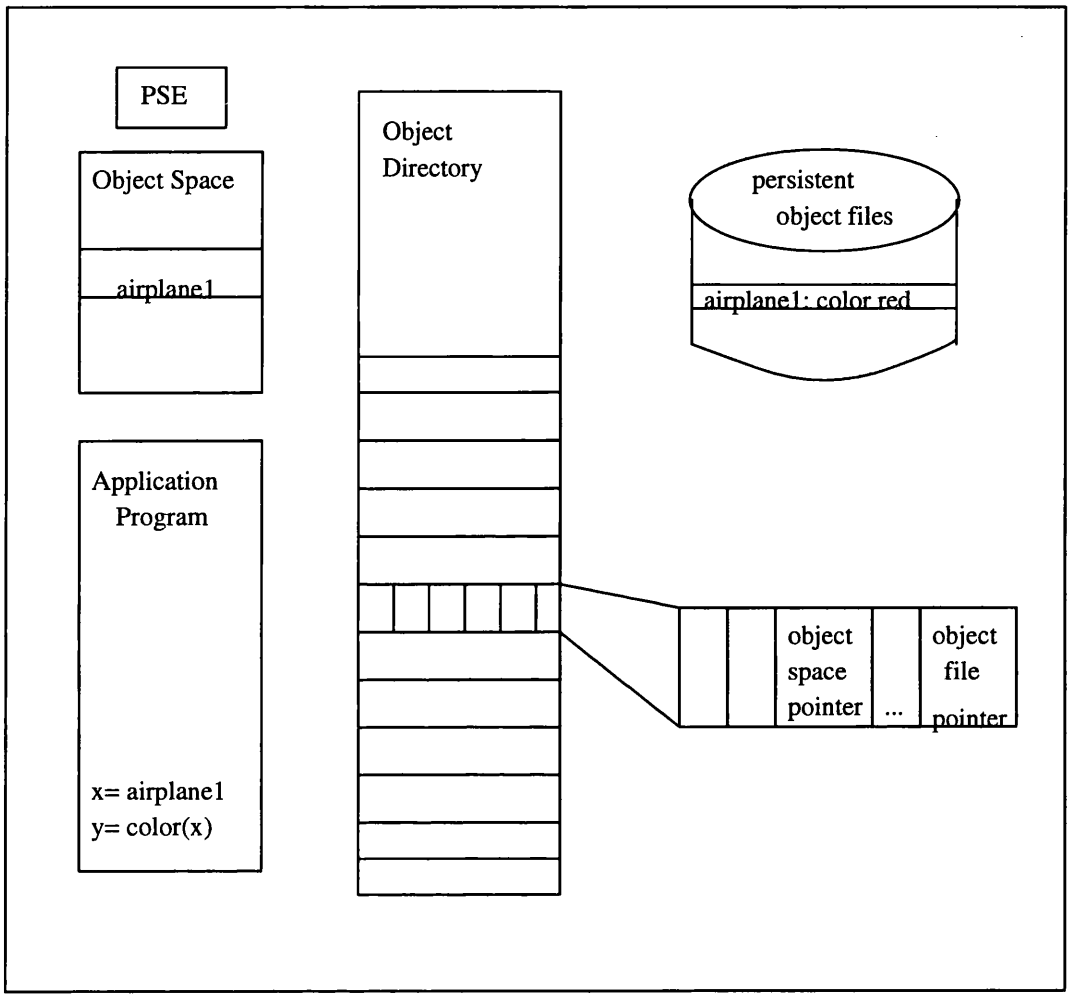


Figure 2-1: Components of PSE

object file pointer	object memory pointer	update mode	object id	object size (bytes)	loaded	local slots	in db	modified

Figure 2-2: Contents Of A Persistent Object Handle

upon modification. If it is *deferred-update*, the object will be updated when the number of objects in the object space reaches capacity thereby triggering garbage collection of the object directory and updating of necessary objects. *Local-copy* objects only exist in main memory and therefore are not updated on disk. the *local-slots* field in a handle exits to store pointers which can not be represented in the database in a persistent manner. Other fields include *indb* (flag indicating if the object is in the database, *modified* whether object has been modified by the current program, and *in-memory* – indicates if the object is loaded into the Lisp system or not. These flags are stored in a bit vector in the handle to minimize their memory consumption.

Each object is stored as a fixed-sized record. If an object is modified to increase its size such that it exceeds the fixed size allocated, then the object is moved to the end of the file. Its size allocation is then increased to meet its new specification. The system also contains a routine that will do *database* garbage collection of the unreferenced objects residing in the database. Unreferenced database objects result when they get moved to the end of the file, because their updated size exceeds the fixed size allocated to each object.

During program execution, object handles are used as parameters to represent simulation objects. When a slot in an object is referenced, one of two actions is taken: if it is determined that the object is not in primary memory, then it is fetched and instantiated before the slot value is returned. Alternatively, if the object is already in primary memory, the value of the slot is simply returned. Also when a persistent slot value is modified in a program, the underlying persistent object system propagates the new value to secondary memory transparently. If the object whose slot value is being modified is not currently instantiated, then the persistent object system will fetch and instantiate the object before modifying the slot value.

2.4.1 PSE System Parameters

PSE's persistent object system includes a set of parameters that either report on the state of the system or can be modified by the user to tune performance and to measure the system's behaviour. The parameter **memory-full** is a global variable that indicates the the *maximum number* of objects that the system will allow in memory (or object space) before garbage collecting the object directory. Since the number of objects required to swamp a virtual memory system varies from system to system and is dependent on the size of the objects as well, **memory-full** is a variable that the user can set to tune the system to his or her configuration. Another useful parameter, **instance-count**, is set by the system and indicates how many persistent objects are currently in the system. The variable **object-faults** is also set by the persistent object system and records the number of times any object was requested by an application but was not in primary memory and therefore, needed to be read and instantiated by the system. Finally, **directory-size** is set by the user and is the size of the object directory. If a larger directory structure is needed due to the creation of persistent objects, the system will dynamically double the size of the object directory. The combination of these system parameters with the three choices of update modes, provides users with facilities for comparing performance under different PSE system constraints.

2.5 Port of PSE to EuLisp

Concurrency provides the possibility of reducing the real execution time of a program through simultaneous execution of different code segments. PSE was ported to EuLisp as a part of the work for this thesis, because it provides a thread facility on the Stardent Titan multiprocessor which allows the implementation of concurrent programs. The components ported to EuLisp include the persistent object system (just described) and the discrete-event and process-based simulation utilities described in Chapter 4 of this thesis. Those were the components implemented at RAND in Common Lisp. The port was necessary, so that the extensions described in this thesis could be undertaken.

The port of PSE to EuLisp required a significant amount of time, because the entire interface with the object system had to be rewritten, and EuLisp modules provided severe restrictions. In the Common Lisp version of PSE, the object system had been merged with the file system through low-level modifications. A new EuLisp metaclass was created for

persistent classes and objects and methods were added at the low-level to handle access and modification of these objects.

Persistent classes were the greatest problem to overcome. In Common Lisp, when a class was input from the database, PSE would construct a call to *defclass* and *eval* to instantiate the class. This technique was not possible in EuLisp, because it does not allow *eval*'s. Another problem is that when a class is created, the accessors for the slot values need to be defined. In Common Lisp, *defclass* defines the accessors. However, in EuLisp, the lack of *eval* makes it impossible to construct and execute a call to *defclass* as in Common Lisp. *Make-instance* on the metaclass can be used to create a class, but *make-instance* doesn't create accessors. The accessors then need to be created manually. A problem arises when creating the accessors in EuLisp at run time, because the accessor functions must get bound to the accessor names, but EuLisp does not allow this kind of dynamic binding to take place. This problem was only partially solved by having the accessor names defined at system load time. This restriction made the EuLisp version of PSE less seamless or transparent to the programmer than the Common Lisp version. Also, it required more than twice as much code to implement persistent classes in EuLisp than it did in Common Lisp.

Likewise, module restrictions made it impossible to have applications reside in separate modules from the PSE module. Since EuLisp does not allow mutually referential modules, it was necessary to define all persistent classes in the PSE module, because local bindings can only be done in macro expansions and the functions to build the accessors are in the PSE module. Therefore, since the accessors were defined in the PSE module, the application must be a part of the PSE module if it wants to use the slot accessors.

There are some compromises that can be made to solve this problem. The first compromise would be to store classes in their own module, then all of the slot accessors would have to be declared in the export list of the module. This solution is quite unsatisfactory though, because when one imports a module of classes, it will load all of the classes in that module instead of loading classes as demanded by the program. Also, it requires the application programmer to structure his or her code differently from how he or she would if the classes were non-persistent. This requirement violates the definition of persistent systems having a *seamless* interface between the program and the database. Another solution, which was implemented, was to provide a construct called *persistent-classes.* This construct is used when loading classes from an already existing database. When persistent classes are defined, the system stores code in the database which when loaded into EuLisp and executed

will create the accessors for a class and bind them to the slot-names listed in the call to *persistent-classes.* Thus, the use of the *persistent-classes* construct does create a seam of sorts between the program and database, but it's less of a seam than would be required if the classes were required to be in a different module.

On a more positive note, EuLisp slot descriptions provided an elegant solution to the problem of access and modification of persistent slots. A macro called *defdbclass* was defined which spliced in the *slot-class* to be *persistent-slot-class.* Then, a slot access method was defined on *persistent-slot-class* to handle the specific mechanics of access and modification of a persistent slot as described previously.

The remaining elements of the port was spent dealing with technical differences between EuLisp and Common Lisp of which there are many but these are not of great interest, so they will not be discussed further.

In the next section, the results of a performance analysis of replacement strategies for objects in primary memory is presented. It was undertaken to determine which replacement algorithm would best improve the system's performance. It is important to make the sequential code efficient to avoid defeating the purpose of parallelizations described in later chapters.

Chapter 3

Performance Analysis of Replacement Algorithms in a Persistent Object Cache

Since the improvement of performance through parallelism is one of the goals of this thesis, it makes sense to be certain that the persistent object system component be as efficient as possible. Thus the motivation for this chapter is to improve the efficiency of PSE's persistent object system which in later chapters will be merged with parallelism to support various simulation paradigms.

The problem of effective object replacement is important, because when using a system with tens of thousands of objects, loading all of them can cause the virtual image to reach a size where it thrashes the virtual memory system. Therefore it can be advantageous to have a mechanism that limits the number of objects in primary memory and manages their replacement. This performance analysis was done to improve the efficiency of PSE by finding a sufficiently good object replacement strategy.

3.1 Related Work

A lot of work has been done on page-replacement algorithms in virtual memory systems. These studies are similar in terms of determining which segments of code or data to replace in real memory, but they are different in that they deal with fixed-size pages. The study in this chapter is concerned with objects which contain a semantically-determined amount of

data and no code.

A previous study was done to investigate the advantages of dynamic grouping of persistent objects using the virtual memory grouping mechanism in Smalltalk [Wil87]. Four models were examined:

1. Near-Optimal: a memory trace was used to determine the sequence of memory references and to assign groupings appropriately.

2. LRU: objects were grouped together based on their frequency of use.

3. Random: objects were assigned to groups randomly.

4. Static: objects assigned to groups before the application began execution, and they stayed in their assigned group throughout the execution.

The study found that under different grouping schemes and page sizes tested over 15 different memory sizes that dynamic LRU provided the best results in reducing the number of page faults. While this related study does in no way prove what is the best technique to use under all circumstances, it does provide a point of reference which can be used for developers of persistent object systems.

Another related study was done which compares various types of caches [Wil90]. It compares various address translation techniques which can improve the performance of cache lookup. It simulates algorithms using an associative memory cache to be implemented in hardware. This approach is different from the study reported in this chapter, because it relies on special-purpose hardware (a hardware cache for objects), whereas the study in this chapter is geared for general-purpose hardware.

Another related work was a performance analysis on PS-Algol [Bai89]. It measured instruction execution speed, the number of instructions executed between jumps, and the number of instructions executed between object faults. The results of relevance to this paper were that programs began by initialising a working set of objects, they then used the working set during program execution, and finally created a fresh working set for program termination. This result provides a strong argument in favor of the caching of objects in primary memory, because the cache can store the objects in the working set, thus reducing object faults.

Finally, there is related work [Koc90] that covers how to maintain a consistent cache in a distributed environment. However, it is concerned with the integrity of caches rather than

replacement algorithms.

3.2 Differences between this study and related work

The applications used in this chapter apply to terrain-based and activity network simulation models and therefore the results are of use to a more specialized community. However, there are fundamental differences with the related work of the previous section that make this study worthwhile:

1. The objects are different: in Smalltalk, nearly everything is an object. In this study, we use a system in which only significant data items are objects (ie. not a pure object-oriented system). Road and activity network objects are used. Road objects consist of roads, intersections, bridges, and deadends. Each road object has several attributes and has an average size of roughly 500 bytes. The activity network objects average 64k bytes due to each one having a stack group to execute in.

2. PSE does not rely on the virtual memory system. Each individual object is treated as a segment and faults occur on objects not on groups of objects.

3. This study does experimentation with fetching of objects within applications.

4. This study tests extends the Smalltalk study mentioned in the previous section [Wil87] by testing new algorithms designed by the author.

5. The data collected was generated using an actual working persistent object system (PSE) rather than a simulation of one as was done by Williams et. al. [Wil87].

3.3 PSE's Caching Mechanism

Figure 3-1 illustrates the object directory and the cache. Handles are used to store status information on the objects they refer to. Handles for all of the objects existing in the currently opened database reside in the object directory in primary memory, from the time when the database is opened until it is closed. The cache contains the handles of the objects that are currently in primary memory. The caching mechanism determines which objects to remove from primary memory when the loading of a new object results in cache overflow. The *cleaning* of the cache is the process of determining which object or

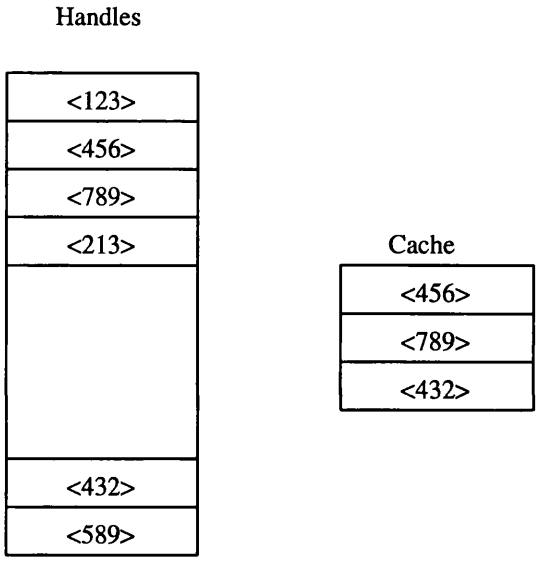


Figure 3-1: PSE Cache Structure

objects to remove and then removing them. After the cache is cleaned, removed objects get garbage collected. The replacement algorithm scans the cache and determines which objects to removing based on a maximum or minimum count depending on the algorithm. I conducted some experiments to determine how many objects to remove from the cache each time it was cleaned. I experimented with removing one, two, and three objects at a time, and found that removing two produced the best results. Thus, I decided to stick with removing two objects each time the cache gets cleaned.

PSE stores handles in a two-tiered structure (see Figure 3-1): it has a directory which contains all of the handles for the objects stored in the opened database, and it has a linked list, referred to as the cache, which contains all the handles whose corresponding objects currently reside in primary memory. The cache exists to shorten the search of handles when cleaning occurs. The cache has a fixed size which is set by the user or by system default. It should be set below the level where the number of objects swamp the virtual memory system. This value depends on the size of the objects and the amount of primary memory. In Figure 3-1, handles <456>, <789>, and <432> are listed in the cache and thus point to objects that reside in primary memory. The remaining handles in the handle table point to objects that exist only in secondary memory and not primary memory.

In a PSE application, the cache will be cleaned when it is full and a new object fault occurs. An object fault refers to the occurence of a slot access or modification by the

application program to an object not currently residing in primary memory. The slot access or modification causes the persistent object system to input the object from secondary to primary memory.

3.4 Caching Techniques

The caching techniques examined consist of those used in virtual memory systems [Lis79] and some original techniques developed by the author. It should be noted that it is likely that the behavior of the persistent object system will differ from that of a virtual memory system, because objects are essentially data, and the working set in a virtual memory system corresponds to the combination of code and data in a program. One can imagine pages in the code section of a program needing to be accessed more than data throughout the course of execution due to loops and repeatedly called subroutines. This process may occur repeatedly. Data, on the other hand, is more likely to be used and discarded. Once a data item has been referenced, it may not be needed again.

The following list describes the techniques investigated in this chapter.

1. Least Recently Used (LRU): replace the object that has been least recently used. LRU is widely considered to generally be the best page-replacement algorithm for virtual memory systems.

2. Least Frequently Used (LFU): replace the object which has been used least frequently since the last time the cache was cleaned.

3. First-In First-Out (FIFO): replace the object which has been resident the longest.

4. Programmer determined: have a construct that can be used by the application programmer to throw out an object.

5. (Faults Out): replace objects that have faulted the most based on the heuristic that the more an object has been loaded into the cache, the less likely it will be needed again. The fault count is reset to zero for each object each time a new application is initialised. This algorithm differs from LFU and LRU in that the count is over the entire execution of the program. In LFU and LRU the count is base on accesses since the object was loaded and is reset to zero when the object is reloaded.

6. (Faults + Acc): add the number of faults and the number of times accessed. Replace the object with the highest combination of values. This algorithm assumes that the more an object has been faulted and accessed, the less it will be needed. As with *Faults Out* the fault count is set to zero for each object when a new application is initialised.

The reasoning behind the usage of the fault counting heuristics is related to the behavior of the algorithms used in this evaluation. It is thus important to terrain-based and activity network simulations. Kruskal's and Dijkstra's algorithms traverse the nodes in the network. Once these algorithms have calculated the distances to and from a node, it is less likely that the node will be accessed as much in the future. Thus, the heuristic states that *the more an object has faulted the less likely it will be referenced.*

An audit trail of the behavior of the cache was generated to look for patterns that might lead to the development of more algorithms. Careful examination of the audit trail information after many executions of different applications on varying data sets failed to show any clear patterns. Thus, the algorithms are based on heuristics.

3.5 Application Test-bed

The following applications have been used as a test-bed for the different caching techniques:

- *Dijkstra's Shortest Path Algorithm*[Gou88]: finds the shortest path from a given node to all the other nodes in the network. Executes on a data set consisting of road, intersection, bridge, and deadend objects. Two versions are used:

  1. Standard algorithm.
  2. Object removal version that uses functions to remove objects explicitly from the cache when no longer needed by the algorithm – if we could do this optimally, the rest wouldn't be necessary.

- *Kruskal's Traveling Salesman Algorithm*[Gou88]: produces a sub-optimal solution of the shortest path that visits all the nodes in a connected network. It uses the same set of objects as does the shortest path algorithm. Its behavior is similar to Dijkstra's algorithm, except that it traverses the road network differently and produces a different number of object faults. The standard algorithm is used as is one that does explicit removal of objects from the cache.

- *Activity Network simulation:* consists of two applications that simulate different process-based activity network simulation models. Activity network simulation models have two basic components: activities and resources. Each activity must have certain resource(s) available to begin execution. Thus, an activity must sometimes suspend execution while waiting for its required resource(s) to be freed by another activity. Once the required resources are available, the activity places a lock on them and processes for a given amount of simulation time. The activity network applications utilize PSE's process-based simulation facilities [Cam91].

The result is six programs which make up the application test-bed. The test-bed is practical for simulation applications: the shortest path and traveling salesman algorithms are commonly used on road network flow simulations, and the activity networks are commonly used to model the requirements for servicing and maintaining equipment.

3.6 Results

The timings were executed on a Stardent Titan III with the EuLisp [Pad91] version of PSE. Figure 3-2 displays a graph of the number of object faults generated for Dijkstra's shortest path algorithm. LRU works best for the medium and larger sized caches, but Faults Out gives the best results when the cache is small.

Figure 3-3 displays a graph of the number of object faults generated under Dijkstra's algorithm under explicit object removal. It produces suprisingly similar results to the version without explicit object removal.

Figure 3-4 covers the number of faults for Kruskal's traveling salesman algorithm. Faults + Acc and Faults Out produce significantly better results than the other techniques. Since the other techniques have similar results, they are represented as a single graph the points of which are denoted by ♣. Likewise Figure 3-5 illustrates the object faults for Kruskal's algorithm with explicit object removal. The results are also quite similar to the non-explicit removal version. The only real difference is with the "others" which do a little better with explicit object removal.

Figures 3-6 and 3-7 display the results for the medium and large activity networks. The large activity network has several methods that produce similarly good results with the slight edge being given to LRU. The results for the medium activity network show Faults Out being best for larger caches and LRU being better for smaller ones.

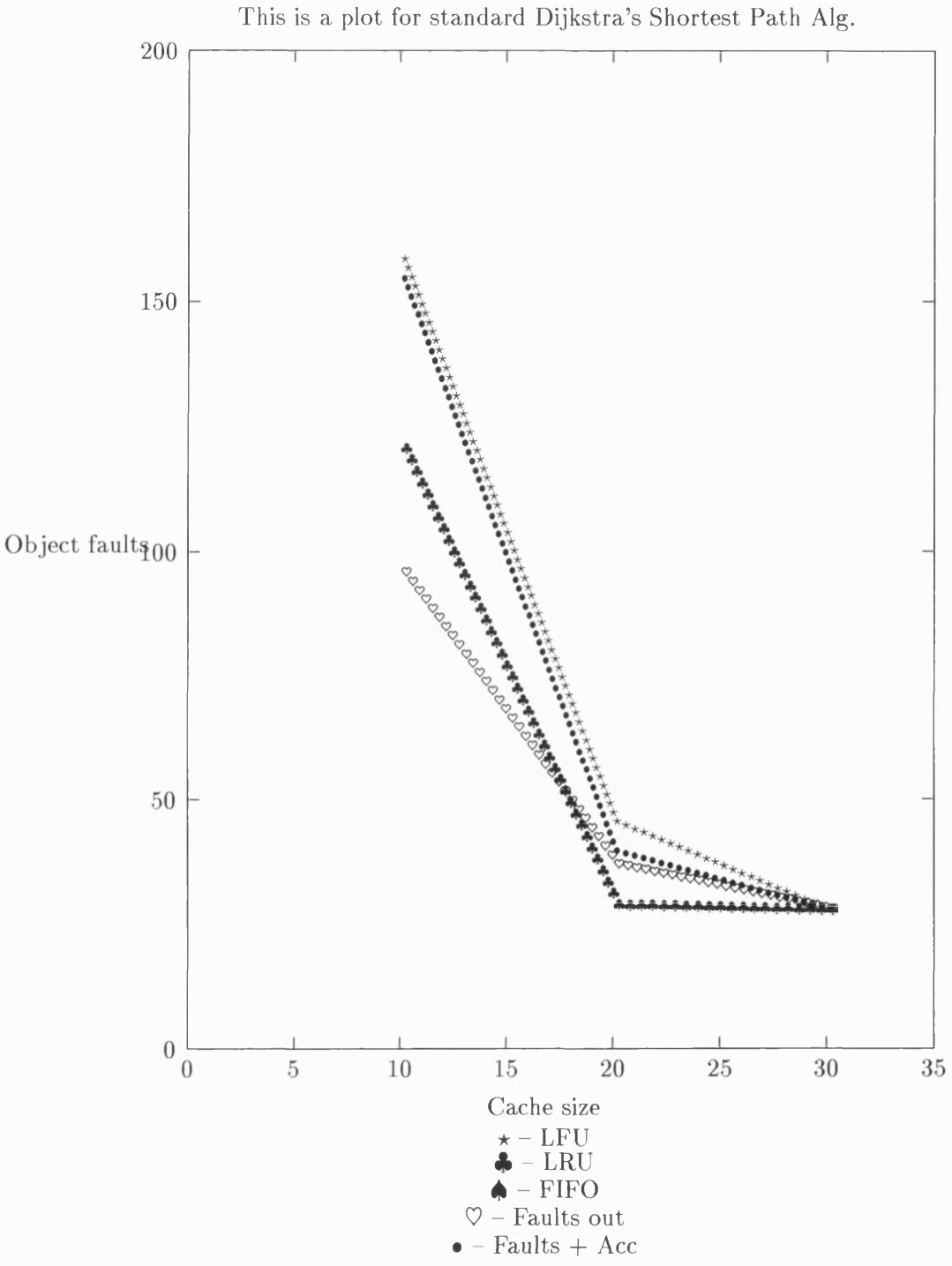


Figure 3-2: Dijkstra's shortest path algorithm

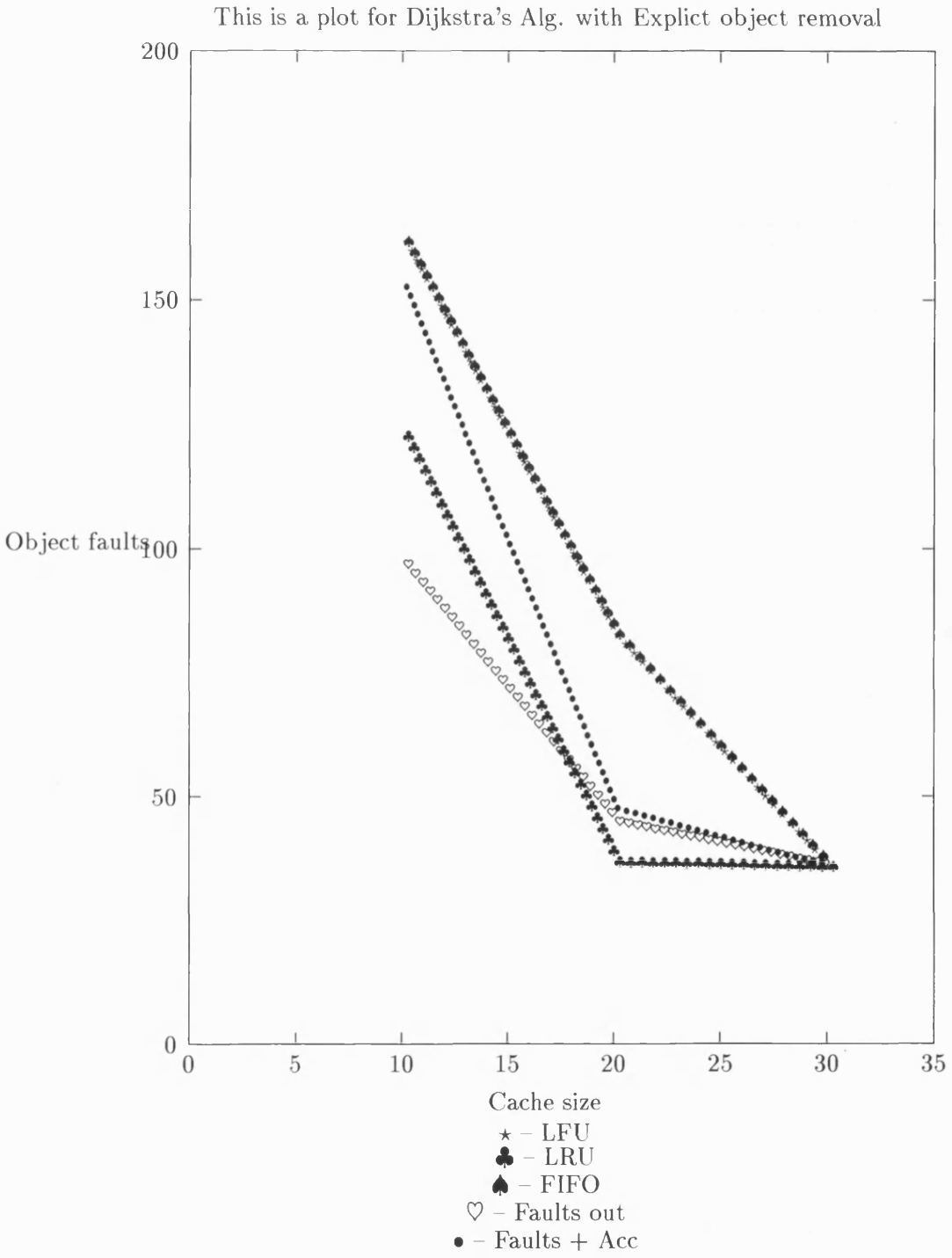


Figure 3-3: Dijkstra's shortest path algorithm with explicit object removal

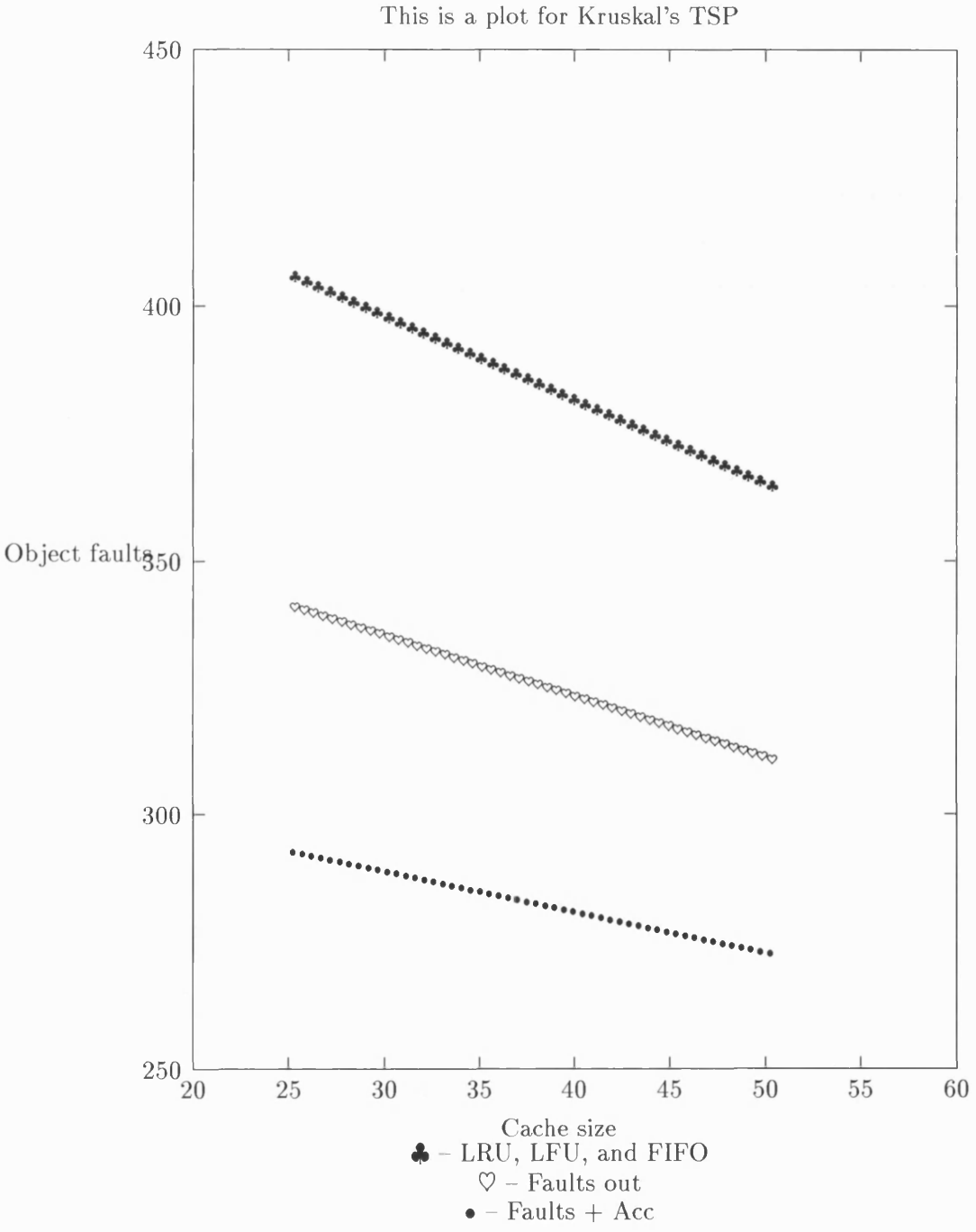


Figure 3-4: Kruskal's traveling salesman algorithm

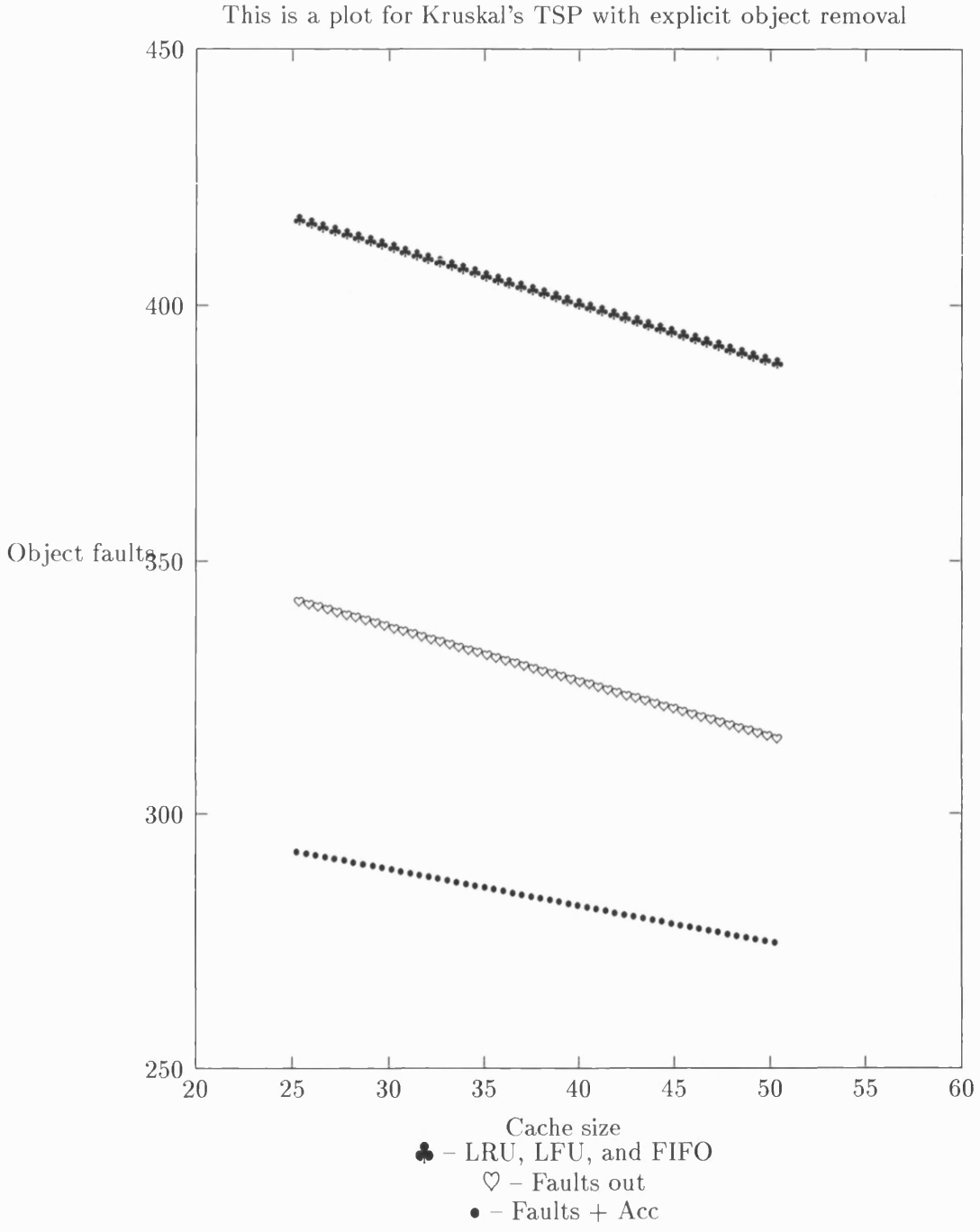


Figure 3-5: Kruskal's traveling salesman alg. with explicit object removal

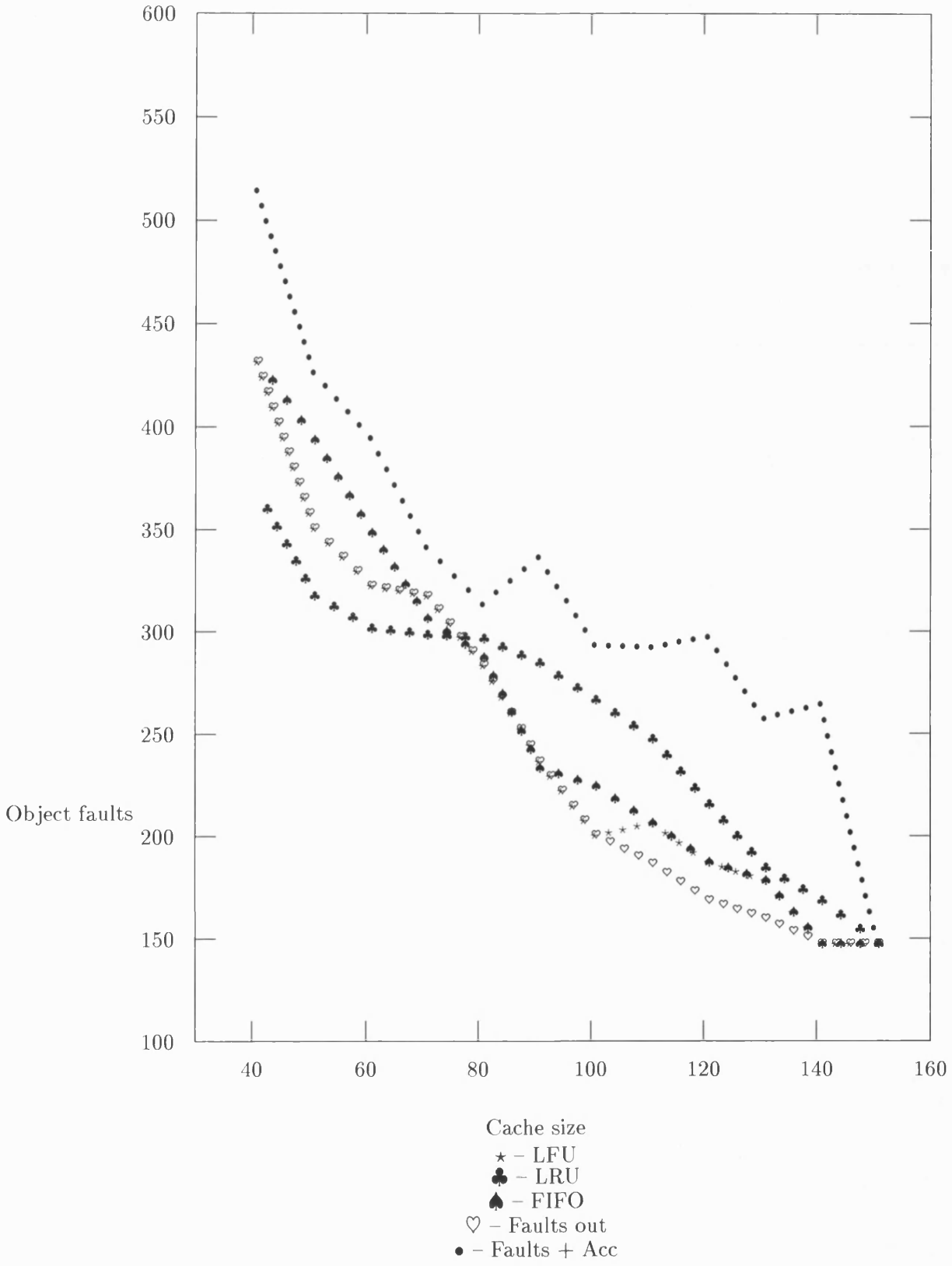


Figure 3-6: Large Activity Network

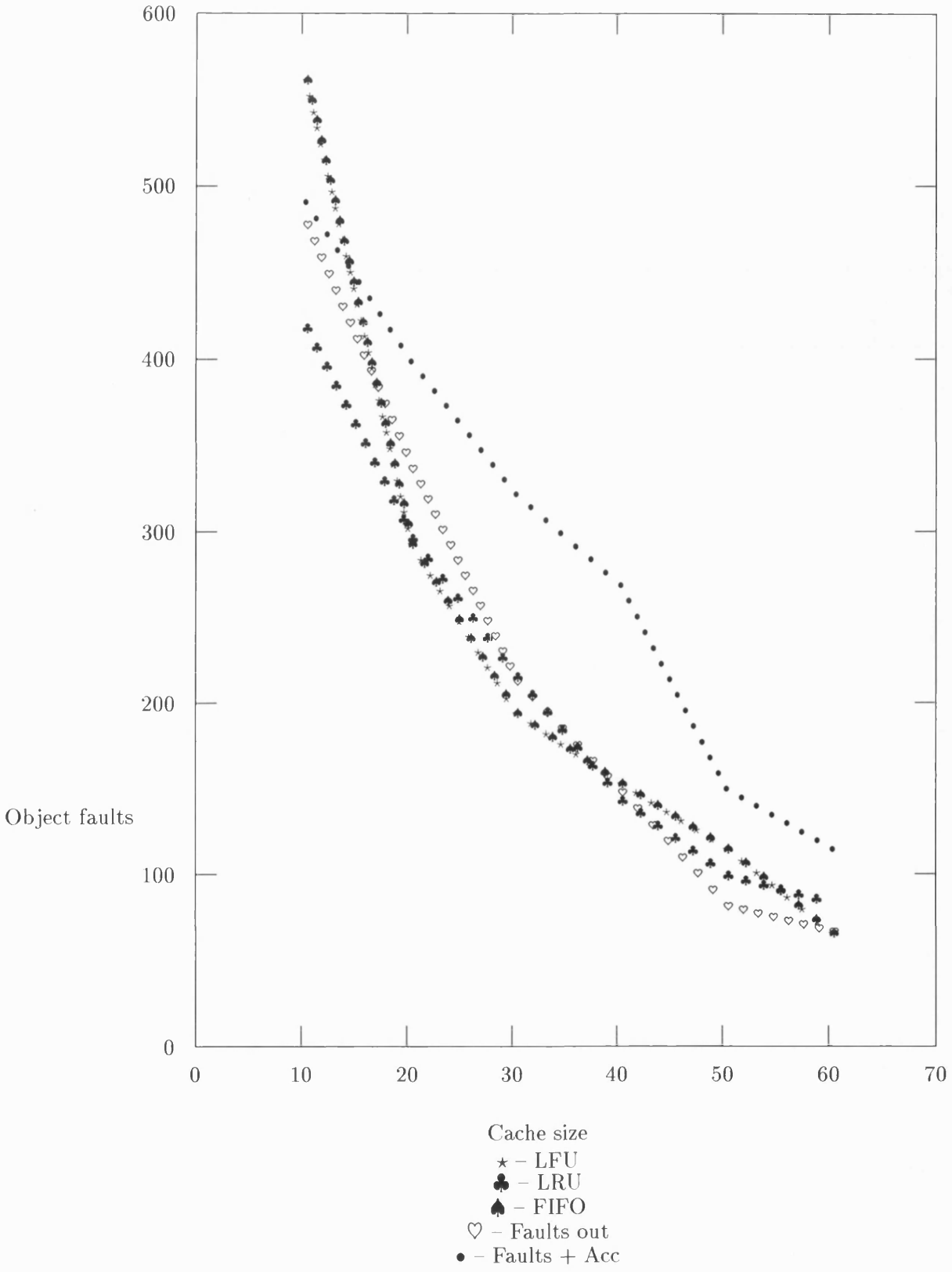


Figure 3-7: Medium Activity Network

3.7 Discussion

The results indicate that Faults Out and Faults + Acc. (both developed by the author) perform better or comparable to the others. However these examples have only been shown to occur under the applications used in this study. There may be different results under other simulations. The results are contrary to those found under operating systems which point to LRU as the best for reducing page faults. It is likely that the reason that performance is better for Faults Out and Faults + Acc is due to the nature of the objects. Herein lies the most important result of this study, LRU can be improved upon by using techniques (such as Faults Out and Faults + Acc) which are designed with knowledge of the domain. There is no code being swapped–only data, and as a result, the behavior of the persistent object system is different than that of operating systems.

LRU does, however, appear to be a good conservative choice, because it usually does quite well and in some cases performs best. One approach would be to do an experiment for a specific application to determine which technique works best in each particular situation.

The results also show that there is no real improvements gained by having programmed the explicit removal of objects from the cache in an application. The poor performance of explicit removal of objects is due to the fact that the removal algorithm was not optimal, because programming an optimal removal algorithm was found to be generally difficult, if not impossible. Therefore, these experiments indicate that it is better to use the object removal heuristic that works best and forget about the explicit removal of objects.

3.8 Conclusions

Caches are important in a persistent object system to minimize the amount of persistent objects stored in primary memory at any given time, to allow object sharing, and to avoid swamping the virtual memory system. The comparison made in this chapter is between standard page replacement algorithms used in operating systems and original techniques devised specifically for persistent object systems.

Experimental evidence shows that it is more efficient to remove two objects from the cache each time it is cleaned. Also, under the applications used in this evaluation, Faults Out and Faults + Acc. perform better than LRU in some cases which is contrary to the current experience in operating systems which points to LRU [Lis79]. Thus, an important

result of this study is that one can do better than LRU by designing replacement algorithms based on domain knowledge. The results show that it may be worthwhile under a persistent object system for a user to try these different techniques to determine which one works best for his or her specific application.

For the purposes of making PSE more efficient, the Faults Out method has been chosen, because it is the most simple and efficient. The determination of using the best caching algorithm is consistent with one of the goals of this thesis which is to improve the performance of PSE. Also, since the persistent object system is merged with parallelism under various paradigms to improve performance, it makes sense to have the persistent object system be as efficient as possible. In the next chapter, a description of the basic simulation mechanisms in PSE is presented. Subsequent chapters present extensions to PSE's simulation capabilities which provide support for various paradigms and hybrids and the merging of parallelism and persistence.

Chapter 4

Object-Oriented Simulation

The two previous chapters of this thesis were concerned with the persistent object system and its implementation. The discussion now moves in this chapter to the basic utilities for object-oriented simulation which were ported from the Common Lisp version of PSE to EuLisp. Utilities similar to the ones covered in this chapter are available in most discrete-event simulation languages. However, it is important to include them, because they are used as building blocks for the merging of parallelism and persistent objects in support of process-based simulation as will be presented in Chapter 5. The advantage of using these primitives in PSE as opposed to some other simulation language is that in PSE they can be used with persistent objects. The advantage of having persistent objects in a simulation environment, in addition to those listed in section 1.1, is that it gives the user the ability to do further analysis of the objects' stored simulation results following program termination.

Object modelling allows us to define a simulation as a set of classes of objects. Each class has attributes which account for its characteristics and procedures which define its behaviour. The behaviours represent the simulation actions taken by or upon the simulation objects. For example, an airplane with its speed, direction, and location attributes can have an *increase-air-speed* operation which when activated will increase the value assigned to the airplane's speed attribute. Once the classes are defined, we instantiate the classes to create objects which represent the environment to simulate. These objects have the attributes and behaviour of their classes (eg. Class Boeing 747 with instance British Airways flight 87). Classes then define a model base which represents the simulation.

As was mentioned in section 1.1, there are three concepts which characterize object-oriented languages:

1. Encapsulation: behaviours or generic functions form the only possible interface to communicate with objects.

2. Polymorphism: a behaviour or generic function can be defined to do different operations for each class or set of classes it is defined for.

3. Inheritance: a class can inherit attributes and operations from another class.

Object-oriented simulation [Bou92], as mentioned in section 1.1, consists of first forming classes of objects with common attributes and behaviours. Classes can refer to real object categories (workstation, waiting queue) or abstract object categories (operation, routing). The modelling governs the programming of the system, so it must be accurate, though it is usually updated during development using stepwise refinement. Object-oriented simulation assumes the representation of system knowledge: objects' characteristics, their behaviours, and interactions between them. In object-oriented simulation, methods or generic functions are used to program state transition logics. The state transition logic is the plan that the simulation is to follow throughout its execution. The state transition logic can be realized as events or processes which determine which objects' states get affected. At each processing of an event or process, when we launch a corresponding state transition logic, we activate associated events or processes through associated generic function calls on them. Time stamps are associated with event and process invocation to allow the simulation programmer to specify the ordering of events over time. For example, the behaviour *change-direction* on class *airplane* needs to have a time associated with it, so the simulation can know when in time the airplane is to alter its direction.

4.1 Simulation Capabilities in PSE

The discussion at this point will cover the basic simulation models that PSE supports: event-based and process-based discrete simulation. Subsequent chapters will focus on higher-level models supported by PSE. Events are actions that occur instantaneously; processes are actions which have a time duration and which may or may not consume resources. Events are scheduled programmatically (or by the user) to occur at the current simulation time or at some time in the future. Processes are also scheduled to begin at a certain time; however, depending on the availability of necessary resources and the priorities of

competing processes, the PSE scheduler controls the activation, interruption, reactivation, and termination of processes.

A global clock object maintains the scheduling and processing of events. However, because PSE is CLOS and Telos-based, it takes advantage of generic functions, described in more detail in section 4.1.1. In contrast to message-passing languages like Ross [McA82], which discriminate methods on only a single argument, generic functions allow methods to discriminate on multiple arguments. In addition, PSE contains routines for sampling from normal, Poisson, and exponential probability distributions to facilitate non-deterministic stochastic processing, which is not supported in ROSS.

PSE's process facilities are modeled after those found in Simscript [Rus79] and Simula. Once a process is scheduled, control is turned over to PSE for activating the process. In many cases, processes utilize resources; and, if a required resource is not available, indefinite delays can occur. When the resource is relinquished by another process, it is then assigned to the scheduled process and activation begins. Below the event and process-based simulation capabilities of PSE are discussed in more detail, and some explanatory examples are presented.

4.1.1 Event-based simulation

The event-based simulation primitives discussed in this section allow the simulation programmer to send messages to other objects with time stamps for when they should execute. The scheduling primitives like *do-at* and *do-after* place the event with the timestamp onto the clock's queue of events to execute in timestamp order. As the simulation executes, the clock picks the next event off its queue, updates the time to the timestamp value, and executes the event. The simulation continues executing until all scheduled events are processed.

The code presented in this section is for a simple carwash simulation which contains several stations that cars, to be washed, must pass through. Each station has a queue that cars must wait in before being processed. Therefore, the carwash is a simple queueing simulation. The following method (*add-to-queue*) can be executed as an event, because it has a resource as its first argument.

```
;;;
;;; add an auto to a carwash's input queue.
```

```
;;;
(defmethod add-to-queue ((carwash resource) (object auto))
.
.
  < other functions associated with adding an auto to a carwash queue>
.
.  )
```

The method *add-to-queue* merely adds the *auto* to the carwash resource (eg. vacuum, wash, and wax) for processing. It can be scheduled as an event by the PSE *do-at* function as follows:

```
;;;
;;; schedules the method "add-to-queue" to occur every 10 time units
;;;
(defun run-carwash (carwash list-of-vehicles)
.
.
   (setq wash-time (current-time))
   (dolist (object list-of-vehicles)
           (do-at carwash wash-time '(add-to-queue ,carwash ,object))
           (setq wash-time (+ wash-time 10)))
.
.  )
```

The function *do-at* will add the method *add-to-queue* to the list of scheduled events. Because the first *add-to-queue* event is scheduled for the current time, the scheduler will process the event before the clock advances. Another similar PSE function for scheduling events is *do-after*. The function *do-after* has the same format as *do-at*; however, the time parameter indicates a time in the future relative to the current time.

Telos generic functions give additional modelling power to PSE's simulation facilities not found in message-based simulation languages like ROSS. For example, in ROSS, there can be only one method for *add-to-queue* defined on a resource object. The example below shows how PSE supports additional methods for *add-to-queue* which discriminate on the second parameter *object*. This version of the method is invoked for *add-to-queue* events when the second argument, *object*, is an instance of type *truck*.

```
;;;
;;; add a truck to a carwash's input queue.
;;;
```

```
(defmethod  add-to-queue ((carwash resource) (object truck))
.
.
  < other functions associated with adding a truck to a carwash queue>
.
.  )
```

Thus, through the use of *do-at* and *do-after*, discrete-event simulations can be developed through the scheduling and execution of events on objects. These primitives are basic, but are needed for building the more powerful and higher-level process-based utilities discussed in the next section.

4.1.2 Process-based simulation

Process-based simulation closely models many activities in the real world. Any system where there are clients that require and compete for servers can be modeled in this domain. Examples that map well to process-based simulation include: bank tellers servicing a queue of customers, assembly lines, and maintenance schedules of aircraft. When simulating the specifics of the modelling of, for example, the maintenance of aircraft, the aircraft to be worked on are the clients, and the availibility of technicians and replacement parts are the resources that they must compete over. A process might be *inspect-engine*. It would be invoked on an instance of airplane. The resource to compete over would be the mechanic. If the engine is faulty, then another process would be activated on the airplane to have the engine repaired. The resources it would compete for would be a mechanic and spare parts. Thus, process-based simulation supports a higher level abstraction that maps into many real world systems.

The operational differences between processes and events stem from the definition of a process as an activity that occurs over a duration of time, rather than an event which is instantaneous. Processes, like events, are defined as methods and activated as function calls. However, most processes include a *resource* argument. Resources are declared as a subclass of the built-in class *resource* and therefore inherit methods defining their behaviour within process calls. When a PSE process is activated, the system determines if a required resource is free. If an instance of the necessary resource is available, it is automatically assigned to the active process. Control of the resource then belongs to the process it is assigned to until the process terminates. Scheduling of processes and allocation and deallocation of resources is controlled exclusively by PSE and is transparent to the user and programmer.

In Simscript and Simula, resources must be requested and relinquished by the programmer within the process definition code.

Another feature of PSE processes is the assignment and management of process priorities. Priorities are useful when modelling a scenario with processes of differing precedence. For instance, in a job shop simulation, critical time-dependent tasks should be serviced immediately when they are scheduled. However, lower priority *busy work* tasks can be performed at any time or interrupted if higher priority tasks are waiting. Suppose an active process is utilizing a resource, and subsequently, a higher priority process, requesting the same resource, is scheduled. PSE will suspend the lower priority process, execute the higher priority process, and then resume the suspended process. All process suspension and resumption is managed internally by the PSE system. A user need only specify priorities as an optional argument when defining processes. Simscript and Modsim also support process priorities but require that the simulation application code compare priorities of processes and explicitly suspend processes when necessary. Simula has no built-in capabilities for prioritizing processes. Thus, PSE has an advantage over Simula, because it supports priorities (though only for processes operating on a single resource instance) and handles automatic suspension and resumption of low-priority processes. Also, all PSE process-based simulation utilities support the use of persistent objects which allow the analysis of the simulation data store in objects following program execution. As will be shown in this section, PSE can be used to support processes operating on multiple resource classes though in a rather awkward way which is somewhat of a limitation in comparison to Simula.

Single resource queue vs. multiple resource queues

Two variations of process-based simulation are available in PSE: single queue and multiple queue. Single queue processes utilize a single queue for each class of resource which has been declared. Invoking a process which requires a resource instance results in scheduling the resource request on a queue associated with the class of the resource. When a resource instance of the class becomes available, the system will activate the scheduled process. When the resource is freed, PSE will select the queued process with the highest priority to execute next.

For resource classes with multiple queues, a request by a process is queued directly on an instance of the resource class. The system determines which resource instance to queue the process request by first looking for a free resource and, if none exist, scheduling

the process for the resource instance with the shortest queue. The differences between the implementation code and simulation results for single queue and multiple queue simulations are illustrated below in a simple bank teller simulation.

```
;;;
;;; Code segments for teller simulation comparing single and multiple teller queues
;;;
;;;
;;; Choose one of the following two resource declarations;
;;;
(defresource teller single () ())
;;;(defresource teller multiple () ())

;;;
;;; Define a customer class
;;;
(defclass customer  ()
  ((name :accessor name :initform (gensym))
   (service-time :accessor service-time)))

;;;
;;; Define a "service" process whereby a customer is serviced by a teller
;;;
(defprocess service 1 :resource (tel teller) ((cu customer))
  (work tel 'service (service-time cu)))

;;;
;;; The top level function which creates tellers and customers, schedules
;;; service processes, and executes the teller simulation
;;;
(defun run-teller ()
    (setq *clock* (make-clock))
    (let ((customers nil))
      ((setter get) 'teller 'resources nil)
      (make-resource 'teller)
      (make-resource 'teller)
      (setq customers (cons (make-instance 'customer :service-time 100)
                            customers))
      (setq customers (cons (make-instance 'customer :service-time 10)
                            customers))
      (setq customers (cons (make-instance 'customer :service-time 100)
                            customers))
      (dolist (c customers)
              (process-at 'teller (current-time) `(service ,c)))
    (run *clock*)))
```

In the above code, *defresource* defines a teller resource class. The first argument of

defresource declares the resource class name; the second argument indicates whether the resource is a single or multiple queue resource. The remaining arguments for *defresource* are identical to those for the Telos *defclass* function. The macro *defprocess* defines a simulation process. The first argument passed to *defprocess* is the process name; the second argument of the process definition provides the process priority, and the list following the *:resource* keyword indicates the required resource. The other parameters of *defprocess* are the same as the parameters of the Telos *defmethod* statement. A call to the function *work* within the process definition is used for advancing time during a process. In the function *run-teller*, the code first creates two tellers and three customers with service times of 100, 10, and 100 units respectively. The call to *process-at* for each customer queues three service processes. Finally, *run* puts the clock into motion.

Figure 4-1 shows the results of two versions of the teller simulation: one using a single teller queue and the other with multiple teller queues, one per teller. In the single queue version, the customers are placed on a single queue based on their order of arrival. Customers are removed from the queue and assigned to the first available teller. With multiple queues, customers are assigned to the shortest individual teller queue upon arrival. For the given service times, the single queue version will terminate in 110 time units; the multiple queue version requires 200 units to process all customers. In addition to *process-at* which schedules processes at an absolute time, the analogous function *process-after* schedules processes at a time in the future relative to the current time.

- Single queue simulation utilizes 110 time units.
- Multiple queue simulation utilizes 200 time units.

In all the examples so far, processes have required a single instance of a resource class; however, processes can also be defined without the need for resources using the following functions:

```
(process-without-resources-at <time> '(<process-name> <process-parameters>))
(process-without-resources-after <time> '(<process-name> <process-parameters>))
```

In such a case, the scheduler will execute the process at the scheduled time. No waiting is necessary because no resources need to be assigned to the process.

Multiple resource instances per process

Another unique feature of PSE's process-based simulation utilities, which is not available in either Simscript or Simula, is the ability to schedule processes requiring multiple instances of a single resource class. For example, in a job shop simulation, a work process may require more than one instance of an identical machine tool or other resource. This feature can be utilized only for single queue resource classes and only for processes without a priority parameter. Each process waiting on a resource queue advances through the queue in the same sequence as it was scheduled. A queued process waits until the required number of resource instances is available before it begins processing. When the resources are free, they are assigned to the waiting process and cannot be used or interrupted by other processes. For example, if there is a process requiring ten resources, then it must wait until 10 resources are free before it can execute. A process waiting for less resources is not allowed to jump the queue if they are available. When the process terminates, all resource instances are relinquished and available for use by other processes. The following PSE functions for dispatching a process with multiple resources correspond to *process-at* and *process-after*:

```
(process-mres-at <resource-class> <time> <number-of-resource-instances>
                 '(<process-name> <process-parameters>))

(process-mres-after <resource-class> <time> <number-of-resource-instances>
                    '(<process-name> <process-parameters>))
```

Multiple Resource Classes

The example in section 4.1.2 described the modelling of the maintenance of an aircraft. It mentioned a process called *repair-engine* which compete for both a *mechanic* resource and spare parts resources. The way to model such a system in PSE would be to nest one process inside the other, each one reserves the classes of resource available.

```
(defprocess replace-fan-belt 1 :resource (f fan-belt) ((en engine))
   (work f 'repair-engine-aux (service-time-to-replace-fanbelt en))
   (setq f nil))
```

```
(defprocess repair-engine 1 :resource (me mechanic) ((en engine))
   (process-at 'fan-belt (current-time) '(replace-fan-belt en)))
```

One could argue that this way of using multiple resource classes isn't the same thing as having a single process with multiple resources. While this is a valid criticism, by nesting one process inside as has been shown, the end result is the same as if there was a single resource reserving multiple processes.

Also, in the above example, since once a fan belt is no longer available once it has been used, it shouldn't be freed. Thus the resource or resources associated with the resource variable (*f* in the above example) can be set to *nil* and that way they won't be made available once the process has terminated, because the simulator checks to see if the resource variable has been set to *nil* before it frees any resources. If the resource variable is set to *nil* the simulator won't try to free the resources and since there is no longer a pointer to them, they will get garbage collected by the Lisp system.

Mixed processes and events

Similar to most other simulation languages, PSE supports the combination of processes and events in a single simulation. An example of mixing processes and events is illustrated in the following code which is part of a carwash simulation. The code segment represents the beginning of the simulation when the driver of the automobile pays the attendant for the carwash before the car is queued for washing. The activity of paying the attendant could be modeled by a process that represents the exchange of money, transfer of receipt, etc.; however, since none of these individual activities are critical, the carwash payment is modeled by use of a single event. As the code describes, the driver first pays the attendant and subsequently a carwash process is scheduled. This example also demonstrates the use of stochastic processing by the use of a normal probability distribution for sequencing autos and for the duration of the carwash process.

```
;;;
;;; Before an auto can get washed, the driver must pay the attendant. This
;;; is the method for the event "pay-attendant".
;;;
(defmethod pay-attendant ((dr driver) (au auto))
    ((setter attendant-paid) au (current-time))
```

```
    ;;; After attendant is paid, the car is scheduled for washing
    (process-after 'vacuumer
                   (normal *attendant-delay-mean* *attendant-delay-sd*)
                    '(vacuum ,au)))

;;;
;;; autoinstances is a list of autos to be dispatched for washing
;;;
(let ((start 0))
      (dolist (auto autoinstances)
      ;;; schedules the "pay-attendant" event
              (do-at (driver auto) start '(pay-attendant ,(driver auto) ,auto))
              ;;; payment of attendant for each auto is time sequenced
      (setq start (+ start (normal *start-mean* *start-sd*))))
      (run *clock*))
```

4.1.3 Recording simulation events and processes in PSE

Collecting and analyzing the results of simulation trials is a critical component of a simulation lifecycle. Most simulation languages have statistics gathering routines which can be included in the simulation application code during implementation. PSE has adopted a different approach by transparently maintaining a database of simulation activities. Every simulation activity, including event dispatching, process activation, process suspension, and resource utilization, is recorded in PSE's activity database. With such a complete audit trail of the simulation's activity, a post-simulation trace can be produced in many different formats. Below are two different formats which can be modified by users to accommodate their own analysis requirements. The first trace is a time-based account of the single queue teller simulation. Note, however, that this trace is not generated during simulation processing; rather, the required data is recorded during the simulation and the trace is recreated by retrieving data from PSE's activity database.

```
Time: 0.0
   process service g392 is scheduled with args (#<customer 42346236>)
   process service g392 is started on #<teller 42325446> with
     args (#<customer 42346236>)
   process service g393 is scheduled with args (#<customer 42345606>)
   process service g393 is started on #<teller 42322436> with
     args (#<customer 42345606>)
   process service g394 is scheduled with args (#<customer 42345156>)

Time: 10.0
```

```
   process service g393 is terminated on #<teller 42322436> with
     args (#<customer 42345606>)
   process service g394 is started on #<teller 42322436> with
     args (#<customer 42345156>)

Time: 100.0
   process service g392 is terminated on #<teller 42325446> with
     args (#<customer 42346236>)

Time: 110.0
   process service g394 is terminated on #<teller 42322463> with
     args (#<customer 42345156>)
```

An alternative trace format presented below is organized by process identifier and process status. For each process that is generated, a set of associated data is recorded. This format provides a different organization of the same data presented above.

```
pid = g392
pname = service
scheduled-time = 0.0
start-time = 0.0
resources = #<teller 42325446>
end-time = 100.0
suspended = nil
work-time = (100)
arguments = (#<customer 42346236>)

pid = g393
pname = service
scheduled-time = 0.0
start-time = 0.0
resources = #<teller 42322436>
end-time = 10.0
suspended = nil
work-time = (10)
arguments = (#<customer 42345606>)

pid = g394
pname = service
scheduled-time = 0.0
start-time = 0.0
resources = #<teller 42276556>
end-time = 110.0
suspended = nil
work-time = (100)
arguments = (#<customer 42345156>)
```

In conclusion, object-oriented simulation allows a model builder to make use of objects and methods to describe the characteristics and behaviour of the system they wish to simulate. PSE has constructs for both event and process-based simulation. The event-based mechanisms allow the scheduling of events on multiple resources, which makes PSE more powerful than counterparts like ROSS. PSE's process-based utilities support resources with both single and multiple request queues. It allows priorities to be assigned to processes, and supports automatic suspension and resumption of low-priority processes which makes it more powerful than Simula, Simscript, and Modsim. It also keeps a audit trail of each process's behaviour during execution as a debugging aid.

The utilities described in this chapter were ported from the Common Lisp version of PSE to EuLisp by the author while at the University of Bath. They are necessary to support the development of utilities that support the merging of persistent objects and parallelism. All these simulation utilities for both discrete-event and process-based simulation allow the use of persistent objects which allows analysis of object slots following simulation termination as well as freeing the programmer from file and database complexities. PSE is the only simulation language that supports persistent objects where slot modification updates the database transparently. Parallelism will be shown in this to improve the performance of simulations.

In the next four chapters, the *utilities* that support these paradigms which were developed by the author while at the University of Bath are described. Applications using these utilities are also presented with results.

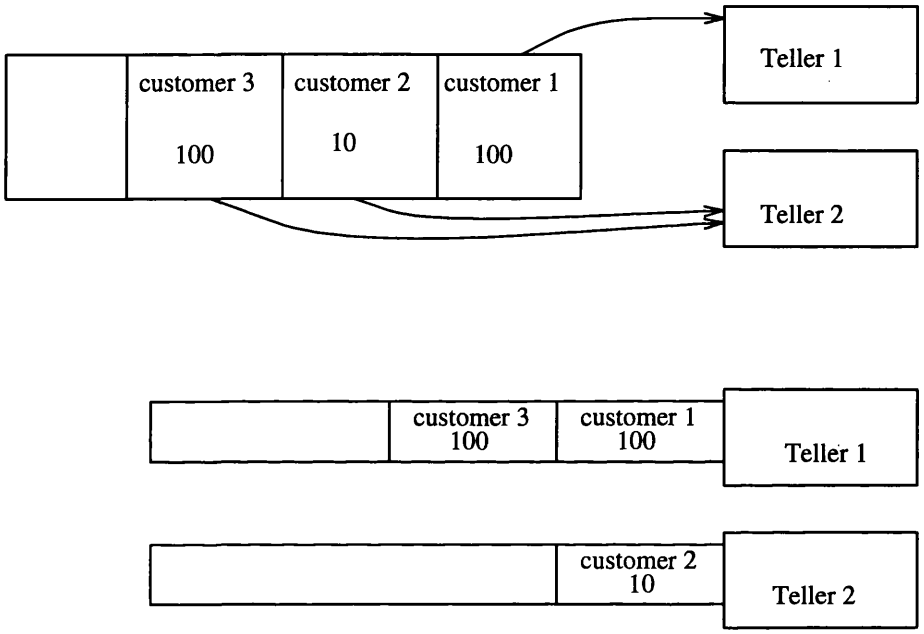


Figure 4-1: Results of two versions of teller simulation

Chapter 5

Concurrent Process-based Simulation with Persistent Objects

In Chapter 3, different replacement algorithms were tested to determine which one of them would make PSE perform the most efficiently. In this chapter, parallelism is utilized as a means of improving the performance of PSE's process-based simulation utilities. The focus in this chapter turns to the use of conservative protocols for parallel process-based simulation in PSE. A comparison between optimistic and conservative techniques will be made with reasons given for the use of conservative protocols. Also, an application will be presented which extends the capabilities of the conservative concurrency control mechanism so that it allows the cloning of simulation objects at run time. At the time it was implemented for this thesis (1991), the ability to add new objects to a simulation was not allowed under conservative protocols. However, I have recently been informed that a group at UCLA headed by R. Bagrodia has developed a system that also does cloning under conservative mechanisms. While I was aware of the existence of Bagrodia's group, I had no knowledge of any work they were doing in cloning. Even so, Bagrodia's group doesn't support persistent objects, so there are no doubt differences between their work and the work presented here.

In this chapter, a technique that makes the dynamic creation of objects under conservative mechanisms possible is presented. An application of dynamic object creation called cloning (where an object is copied) is also presented to show where dynamic object creation can be useful. Cloning is done dynamically to improve the throughput of the system being modelled. It works by cloning simulation objects that cause bottlenecks. Through cloning and the subsequent rerouting of some tasks, the bottlenecks are removed.

Processes are represented as persistent objects which support the perusal of simulation behaviour after the completion of its execution. This chapter also describes why it is advantageous to use conservative protocols to synchronize parallel persistent object-based simulations. An assembly line application is presented which executes under conservative protocols which were modified to permit the cloning of objects.

5.1 Persistent Objects and Concurrency

The motivation for concurrency is to improve the performance of simulations. However, to execute a persistent object-based simulation requires protocols to manage the concurrency to ensure that the simulation semantics are not altered from its sequential version. The choice was made to implement concurrency at the event level rather than the database transaction level, because if dependent events are synchronized on each object, database transactions will be sychronized as well. Event-level sychronization will allow independent events (eg. *move car to station X, process first car on station Y*) to execute in parallel. Dependent events which act on the same object (eg. *move car A to station X, process car A at station X*) will be executed in lock-step. Since events are the parallelizable unit, if they are synchronized based on the write sets of objects (a *write-set* is a group of objects that are dependent on each other because they modify or access the same mutable attributes), so will the database transactions they generate. If dependent events are sychronized, then all database transactions will be serialized for each object, because events are the driving apparatus of the simulation and the only agent which produces database operations. Thus, the environment demands a protocol which will control concurrency for events and the transactions will follow suit.

In choosing a concurrency protocol for PSE, a few techniques were considered:

1. placing semaphores around *write-instance*–which writes an object's current representation to the database.

2. optimistic concurrency control

3. conservative concurrency control

5.1.1 Semaphores

The first approach made for this thesis, to produce concurrent persistent objects, was to place semaphores around PSE's *write-instance* function. The use of semaphores alone, however, did not produce adequate results, because there is more to be synchronized than the writes to the database. Events as well must be synchronized to ensure that the simulation results will be identical to the sequential version by obeying the simulation semantics. Independent events can execute in parallel, but events affecting the same objects must execute in time stamp order. Placing semaphores around *write-instance* does not sychronize events. Therefore, another technique was needed.

5.1.2 Optimistic Concurrency Control

Optimistic and conservative techniques were considered to utilize concurrency at the event-level in PSE. The most commonly referenced optimistic concurrency control mechanism is Time Warp [Jef85b].

The basic device used by Time Warp is to impose a *virtual* time order for every process. This ordering is achieved by having each process queue its incoming messages by time-stamp order rather than arrival order. In this way, a process can be thought of as working along its input queue, increasing its local virtual time (LVT) (the time-stamp of the message currently being handled by a process) to the time-stamp of each message as it gets to it. When a message arrives whose time-stamp is smaller than the process's LVT, the message lands in that part of the queue already processed, thereby causing a rollback.

Time Warp solves the problem of ensuring the correct order of messages by *unsending* them. Each real message has a corresponding antimessage which, when sent to the same destination as the original positive message, serves to annul it. It is crucial that the antimessage carry the same time-stamp as its positive corresponding message, because the arrival of the antimessage must invalidate any work performed by the recipient from that time on, and cause the recipient to rollback to that point if necessary. Should the recipient rollback in response to the antimessage, it will send more antimessages to yet other processes. In this way, the rollback will propagate to the other affected parts of the system as desired.

For this thesis, optimistic methods were not chosen, because of the cost involved with rollback in database systems. If I'd had access to a database system that had a rollback mechanism, it certainly would have made it feasible to do Time Warp rollback using that

mechanism. It seems to me that the implementation of a database system with rollback so that I could make use of it for a Time Warp system is beyond the scope of this thesis. Also, I would personally argue that doing rollback on a database system would be very expensive, due to the slowness of disk operations relative to memory operations. Thus, I would say that database rollback is very expensive compared to rollback implemented in primary memory. The cost of storing persistent objects to secondary memory can require between one hundredth of a second to one second depending on the size of the object and the performance of the database system being used.

The other option for implementing Time Warp with persistent objects would be to require that all modifications to the database be stored in primary memory until the Global Virtual Time (GVT) has passed the transaction's time-stamp which means the modification to secondary memory can then be made safely. However, I would argue that the saving of object modifications in primary memory until GVT is undesirable, because it adds to the already large primary memory requirements associated with optimistic systems [Ung93]. The other large memory usage goes to the state saving queues which record the state of each object from LVT back to GVT, and the message queues, which may be quite large due to antimessages and messages that will be undone.

One of PSE's design goals is to handle large simulations which can require thousands of objects which would likewise require a large amount of primary memory. As a result, the fact that optimistic methods require a large amount of memory for state queues is a negative factor and is one reason for not using optimistic methods with PSE. The other reason for rejection is due the high cost of undoing messages, which result in modifying persistent attributes which, as described previously, is slow and memory intensive under optimistic methods making them unattractive.

5.1.3 Conservative methods

Conservative techniques require that the events for each object execute in time-stamp order. This requirement restricts communicating objects from overtaking one another. Therefore, unlike optimistic methods, there is no rollback.

Conservative techniques can be summarized as follows [Cha81, Fuj90]: If a process contains an unprocessed event E1 with time-stamp T1 and that process can determine that it is impossible for it to receive later another event with time-stamp smaller than T1, then

the process can safely proceed with E1 because it can guarantee that doing so will not later result in a violation of the local causality constraint. Processes containing no safe events must block; this can lead to deadlock situations if appropriate precautions are not taken. The sequence of messages on an input queue must be in nondecreasing order. Each queue has a clock associated with it that represents the time of either the first message on the queue or if there are no messages, the time of the last message processed. Each object repeatedly selects the queue with the smallest time-stamp and processes the message on it if there is one. Otherwise it blocks and waits for a message to arrive on that queue. The protocol guarantees that each process will only execute events in time-stamp order.

Figure 5-1 illustrates the behaviour of a process under conservative mechanisms. The circle corresponds to a logical process (LP) which is represented as an object. It has two input queues that are initialized statically connecting it to the other LPs in the simulation that send messages to it. It has a single output queue it places the output messages on. The LP in figure 5-1 repeatedly checks both IN1 and IN2 for the lower clock value and processes a message on the queue if there is one. Chandy and Misra have shown that by sending null messages, LP's will continue to advance and therefore avoid deadlock. I have chosen Nicol's variation [Fuj90] which only sends null messages on demand as follows: if there is not a message on a queue, the LP will request that the LP on the other end of the queue send a null message timestamped at the sending LP's current time. The receiving LP will then advance the clock value to the timestamp and proceed to find the next queue with the lowest timestamp. This demand-driven scheme was chosen, because it avoids sending null messages unless it is necessary which seems to be more efficient.

It should be noted that there are other techniques for handling deadlock other than the null message technique, such as active null messages. Chandy and Misra [Mis86] have developed a technique that can determine when deadlock will occur, so that corrective measures can be taken without the call for null messages. From the point of view of object cloning, which will be discussed in section 5.3, there is no essential difference between these methods; they all require a statically determined communication graph.

Conservative mechanisms require less primary memory than optimistic ones, because there is no need to save the state each time an event is processed and input queues contain no antimessages. Also, in the case of persistent systems, there is no need to save modifications to the database, because due to the lock step execution of events for each object, once modifications are made, there is no need for the protocol to undo them. While there

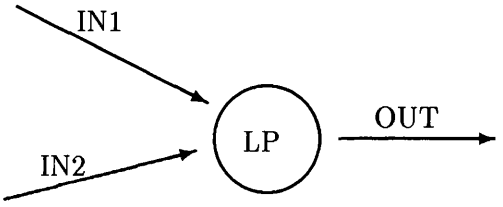


Figure 5-1: A Logical Process

are several techniques for dealing with deadlock in conservative systems[Mis86], the null message technique is chosen because it does not require the temporary shutting down of the simulation that is required by the deadlock detection algorithm.

Livesey presents a combination of optimistic and conservative methods called varimism [Liv90], but it has the same memory demands as the optimistic approach, so it was rejected, because the kind of large-scale simulations that PSE is designed to support contain thousands of objects using large amounts of primary memory. Thus, a memory-efficient technique is necessary. Conservative techniques are memory efficient, because they don't use rollback which requires a large amount of memory-consuming state saving.

5.2 Persistent Objects as Concurrent Processes under Conservative Protocols

The resulting implementation in PSE consists of having simulation processes (as described in chapter 4) represented as persistent objects. Each persistent object is then executed by the system as a logical process with input channels connected to the processes they receive input from. The advantages of representing processes in a conservative mechanism as persistent objects are that each object encapsulates the environment associated with each process, and a history of the events occurring on the process can be stored transparently to the database.

One of the most important advantages of object-oriented programming is encapsulation. It allows a programmer to partition portions of code and data in the program which reduces the overall complexity of the system. In the example to be presented in section 5.3, each assembly line station is represented by an object which contains information about the delay time, stations connected to, thread pointer (each LP gets its own thread to execute in – threads are the programming language construct used in PSE to implement concurrency), and a history list storing the time and name of each job that has passed through the

station. The history list is a persistent slot (as are the connection and delay time slots) which when modified cause the underlying persistent object system to modify the database accordingly in a manner which is transparent to the application programmer. Thus, once the program has terminated, examination of the simulation's behaviour can be carried out through perusal of the history list stored in the database. The thread pointer field is not persistent, because the thread is built at load time and the pointer will vary on each execution.

Figure 5-2 illustrates the data structures used to build the parallel simulation of the assembly line using conservative protocols and persistent objects. The queue structure is supplied by the system and contains a pointer to the message queue, the queue simulation time, the pointer to the semphore created to maintain mutual exclusion on access to it by separate processes, a pointer to the LP object that is placing messages in the queue, and a queue identifier which exists for error checking purposes. The station object as illustrated contains slots for the simulation time of the LP and a pointer to a list of input queues – when cloning occurs new input queues get added to this list. The station object also contains the delay time for the station to work – an application dependent slot. It also contains a pointer to the list of previous station object(s) for error checking purposes. Likewise it has another slot which contains the list of station object(s) it sends output messages to. The station objects are stored in a list to allow for more than one station object to be connected to the other. Finally, it contains a thread pointer and a history list as described earlier. It is important to note that all the slots for the station object must be specified for any other conservative protocol-based simulation under PSE. Extra application-dependent slots may be necessary, however, depending on the application.

The main difference between a sequential and a concurrent process-based PSE simulation from the application programmers point of view is that all simulation objects must contain an extra thread attribute. The thread attribute must be assigned an initial form which consists of a call to the EuLisp *make-thread* function. Likewise the call to *make-thread* must contain embedded code that handles the processing of messages for the logical process object. Currently, the code must be copied for each object with slight object-specific modifications that know about the object's structure. The slots are the same, but the naming may be different, because it is up to the user to define the LP object. It mainly consists of changing the names of the accessors depending on how they are named in the application. A more generic version could be developed, but due to time constraints, it is beyond the scope

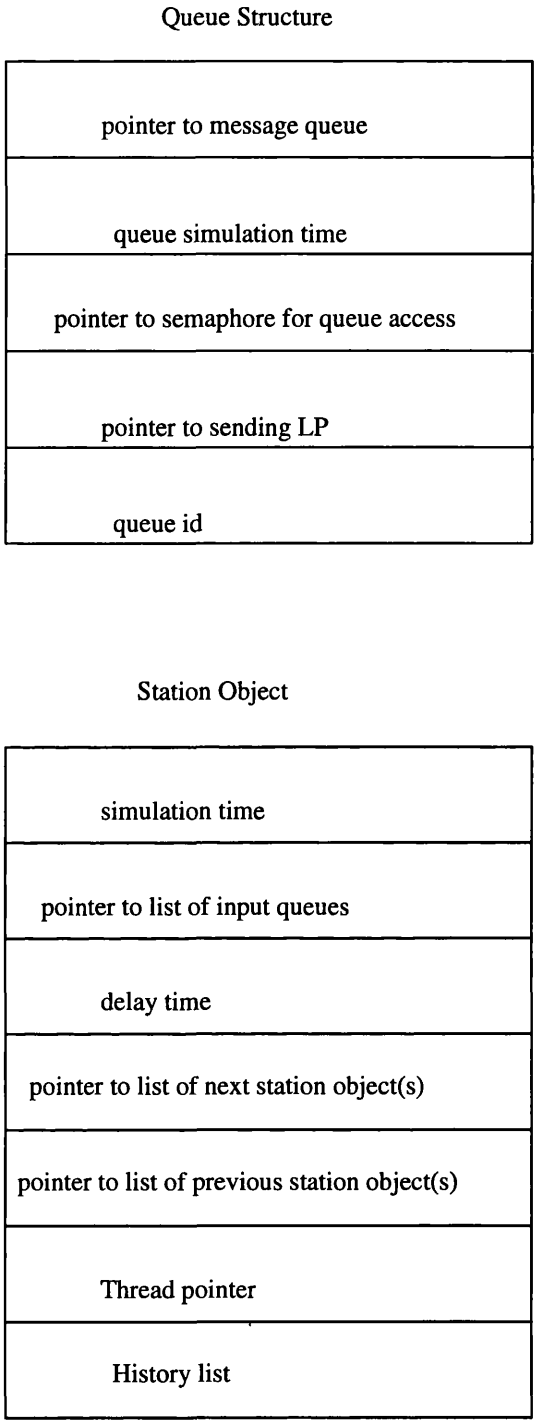


Figure 5-2: Queue Structure and Station Object Contents

of this thesis which is the development of a prototype. Also, extra system code must be loaded into the PSE system which contains utilities that support the conservative protocols as executed in each thread. Thus, the use of the conservative concurrency control facilities in PSE requires some modifications to the user's code.

5.3 Object Cloning

Conservative mechanisms [Mis86] of concurrency control are characterized by a lock-step execution of messages amongst each process. Message queues are initialized statically making it impossible to create new processes dynamically. However, section 5.4 presents a technique that allows processes, as represented by persistent objects, to be added to the conservative mechanism at run time under some circumstances. This method, termed *cloning of objects*, can be useful when developing a model that responds dynamically to its current state; for example, a bank opening another teller window in response to a large queue of customers. Since conservative methods execute events on each object in a lock-step fashion, they typically require pre-knowledge of the connectivity, while on the other hand, optimistic methods use rollback to allow objects to execute events without ensuring that they are executed in time ascending order, and so it is possible to introduce new objects with ease. The development of a cloning mechanism further extends the capabilities of PSE, and the cloning application tests out the mechanism.

Replication of objects has been used in the Time Warp operating system to support dynamic load balancing [Rei90]. Essentially cloning in this context consists of copying an object and its state into another object. This scheme allows new messages to be sent to the clone residing at the new location while the messages sent before the object is cloned can still be processed by the original at the old location.

The motivation in this thesis for cloning objects is different from load balancing; cloning is used here to experiment as a means of eliminating bottlenecks in a simulation model. For a simple example, consider the assembly line simulation described by Misra [Mis86] (see Figure 5-3). Note that it is possible for this model to deadlock under conservative concurrency control given the following scenario: X cannot proceed unless it receives from Z, Z cannot proceed unless it receives from Y, and Y cannot proceed unless it receives from X. This model was implemented using conservative protocols under PSE. The *null messages* technique proved effective in avoiding deadlock.

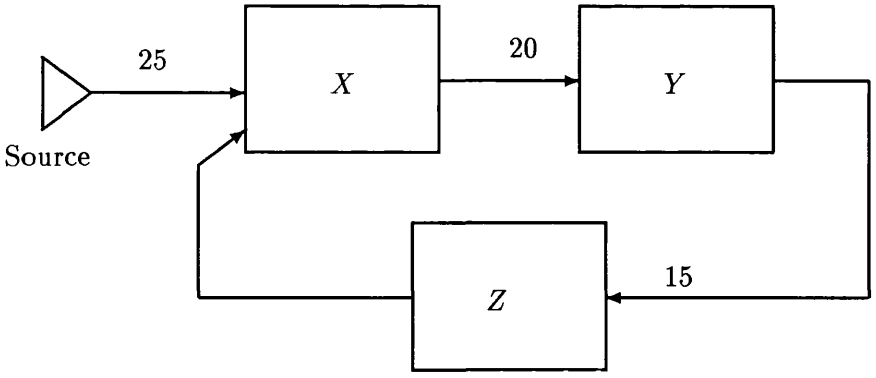


Figure 5-3: Circular Assembly Line

In conditions of sufficient load, it could be desirable to replicate one of the assembly line services, to increase throughput of the model. Thus, cloning can be used to extend the modeling capabilities of the simulation model. It is the method of achieving such a replication which will be addressed in the next section, as well as the implications of the technique for simulation semantics.

5.4 Cloning Technique

The cloning methodology, which further extends PSE's modelling capabilities, works as follows: a candidate is selected if it constitutes a bottleneck in the flow of the simulation model. Often simulations will have a smooth flow of traffic with a few bottlenecks in the simulation model itself which impede its behavior (traffic flow simulations being the classic example). By cloning these objects, bottlenecks can be eliminated to allow a consistent flow throughout the simulation. The *simulator* determines bottlenecks based on the size of an object's input queue. When the number of messages waiting to be serviced on a single object is greater than some α, then a candidate is suitable for cloning.

The cloning problem is solved by copying the contents of an object into a newly created instance of the same class. A symbol must be generated dynamically, so the program will have a means of referencing the clone. After the object is copied, it then must be inserted into the simulation in a manner that will cause it to be utilized. Firstly queues must be dynamically linked to the clone that is connected to the same input objects as the original. Likewise, output queues must be attached to the clone linking it to the original's outputs. Also, code must be added to each object that sends messages to the clone and the original. The code will do a random choice between the object and its clone, so it can determine

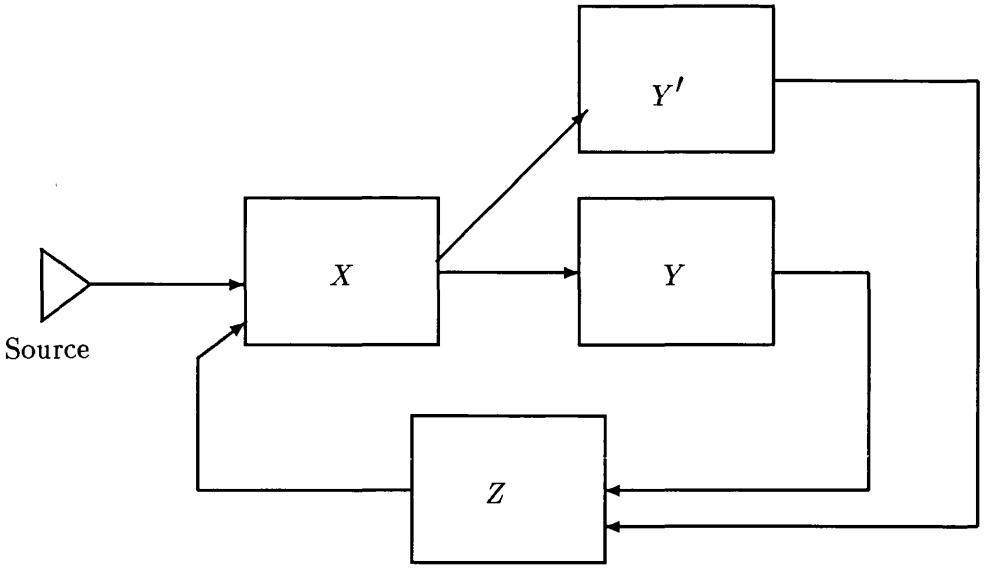


Figure 5-4: Assembly Line Configuration with Clone

which one it should send a message to. This process must be repeated for each input queue that links to the object being cloned. In this way, the number of messages sent to the original object will be halved between it and its clone.

The problem of dynamically modifying the message sending code for objects linked to the original is the biggest hurdle in dynamically creating new objects under conservative protocols. However, due to the simple syntax and the dynamic nature of LISP (the implementation language), the modification of an object's message-sending code is much simpler than it would be in a static language like C.

5.5 Example Clone

A scenario will now be presented where cloning is useful. In Figure 5-3, assume that the delay time on station X is short and long on station Y. As a result, the queue of objects to be processed at station Y will become quite large making station Y a bottleneck in the assembly line. It is therefore advantageous to clone it and divert some of the flow to its clone (Y'). Figure 5-4 illustrates the circular assembly line after station Y has been cloned. There are two queues now exiting from X and there are two queues entering Z. These queues can be added while all processes continue their concurrent execution, because the new queues do not get messages placed on them until after they have been installed.

The following code is a snapshot of output from the circular assembly line simulation where a cloning occurs using the conservative concurrency control technique described ear-

lier. It is executing on a three processor Stardent Titan. During the execution, jobs pass through the assembly line until the size of the queue for station Y gets large, at which time Y is cloned. The cloned station's name, $G00062$, is generated dynamically using a function in the programming language library [Con]. After the cloning occurs, jobs get sent from station X to both stations Y and $G00062$ to reduce the bottleneck. The output is as follows:

```
job c on station y at time 36
job f on station x at time 38
job c on station z at time 44
cloning station
job g on station source at time 40
done cloning station
job a on station y at time 38
job d on station x at time 40
job b on station x at time 42
job f on station y at time 39
job g on station x at time 43
job b on station y at time 43
job g on station y at time 44
job h on station source at time 45
job d on station G00062 at time 41
job e on station x at time 45
job d on station z at time 45
job e on station y at time 46
job c on station x at time 47
job h on station x at time 48
job c on station G00062 at time 48
job h on station G00062 at time 49
job a on station z at time 46
```

Through observation of both the assembly line simulation of Figure 5-3 and the cloning one of Figure 5-4, the queues on Y process gets reduced by approximately one half after the clone. The non-cloning version gets to a state where the Z process spends most of its time waiting for input from the Y process. The Y process develops a large input queue, and the X process likewise mostly waits for input from the Z process.

In the cloning version of the assembly line simulation, it behaves similarly to the non-cloning version, until the Y object gets cloned. After the clone, an equal number of objects get fed to both Y and Y' resulting in double the throughput. As a result, the amount of waiting done by X and Z is dramatically reduced.

5.6 Impact of Cloning on Simulation Semantics

Certainly cloning alters the model of the simulation and therefore the results obtained. Cloning is a means which can possibly improve the behavior of a system being modelled. If the purpose of developing a simulation is to analyze an operation's behaviour so as to better understand it and make it more efficient, then cloning is a reasonable technique that can be built into the simulation that will be triggered by a given set of parameters. Thus, cloning is an automatic means by which the simulation program can improve what it is modelling.

5.7 Future Development

Future work should look into defining whether or not it is appropriate to clone. Also, there are certainly other applications for cloning under conservative protocols that should be investigated. It has been suggested that a simulation of a network of clients and servers would be a good application where dynamic object creation could be used for the simulation of the arrival of new clients.

5.8 Conclusion

A method for cloning persistent objects in a concurrent simulation environment has been presented. The different methods of concurrency control available were discussed. The reason for choosing the conservative approach was explained (it requires significantly less primary memory to execute). The scenario provided found it would be useful to clone persistent objects in the case of handling bottlenecks in the system being modelled. PSE was extended to support conservative concurrency protocols on persistent objects. The mechanism was tested with an assembly line simulation. PSE's support for conservative protocols was then extended to handle cloning. A simulation model that used cloning was then implemented and tested using PSE. Through observation, the use of cloning resulted in reducing the size of the queues on the bottleneck process in the assembly line simulation. The use of cloning therefore increased throughput.

The mechanism used for cloning in this chapter is original and at this point no other known conservative distributed simulation system allows objects to be added to the simulation dynamically as is now available in PSE. The dynamic properties of Lisp lended well to the implementation of dynamic cloning of objects.

This chapter has covered one of the simulation paradigms for which parallelism and persistence have been merged in this thesis. The advantage of representing logical processes in conservative parallel simulation as persistent objects is that it supports the perusal of the simulation's behaviour after its completion and frees the programmer from file and database complexities. While the assembly line example in this chapter is too small of an application to benefit from parallelism, it is useful for presenting a solution to the cloning problem under conservative mechanisms. It also paves the way for larger applications which could benefit from parallelism.

In the next chapter, the discussion will shift to support for connectionist models. Due to the great interest in connectionist modelling, and the number of systems being made available to support it, a connectionist representation language has been incorporated into PSE which utilizes persistence and parallelism to support knowledge based consultation by a simulation. The discussion will cover the language, its capabilities, applications, and performance improvements from mappings onto parallel machines.

Chapter 6

Connectionist Simulations

Up to now, the discussion in this thesis has been concerned with the application of parallelism and persistence to time-based simulation paradigms. Time-based simulation is suitable to modelling a significant amount of real-world problems. However, there are problems where time-based simulation is not suitable. Also, time based simulations, as was shown in section 1.2, often require a knowledge base for consultation on decisions that need to be made during execution. One of the domains where time-based simulation has no application is the modelling of the brain's capacity for inferential reasoning. To support this kind of modelling, PSE has been extended, as a part of this thesis, to support parallelism and persistence for connectionist simulations which support the development of knowledge bases for consultation. Connectionist systems model the reasoning mechanisms of the human brain [Fel82]. Connectionist models can be combined with discrete-event simulations to support expert reasoning where it is necessary. A simple example would be when a moving object needs to make a decision about which road to follow. In such a case, it can be useful for the moving object, whose behaviour is simulated using discrete-event domains, to consult the expert knowledge stored in the connectionist model to make a decision on the proper course to take. Thus, the ability to do both discrete-event and connectionist modelling in PSE supports simulations that need to combine those two paradigms. Likewise, the components discussed in the previous chapters that utilize persistence and parallelism can be used in conjunction with the components described in this chapter as will be shown.

Connectionist models provide a mechanism for representing knowledge through connections between neurons. Those connections are weighted to represent the certainty factors between semantic relationships. Due to the recent increase in interest in the use of con-

nectionist and neural systems, there has been active development in tools that support their development[D'A88, Flo90, Fel88, Wan90]. Thus, to support the increasing demand for connectionist simulators, a component called POCONS (Persistent Object-based CONnectionist Simulator) has been added to PSE. POCONS provides basic primitives for the development of object-oriented connectionist models. PSE represents neurons as objects to support encapsulation, inheritance, and reuse. These neuron objects can be made persistent using PSE's persistent object system.

In this chapter, the POCONS system will be described and examples of its use will be presented. The interesting algorithms required for its implementation will also be described. The following chapter will then discuss applications to parallel computing using POCONS with results from parallel connectionist simulations. It will also present a feature for storage and reuse of neural networks which is a practical application of persistent objects. An extension of POCONS which supports chaotic neural networks will be presented in the next chapter as well.

6.1 POCONS: A Persistent Object-based Connectionist Simulator

POCONS is a new component added to the EuLisp version of PSE which supports persistent object-based connectionist simulation. Other connectionist simulation tools are powerful and expressive, but with the exception of Neula [Flo90] and NSL [Wei91], they do not support an object-oriented design methodology. Both Neula and NSL have object-oriented constructs, but in both cases, their syntax and semantics are unlike the widely used object-oriented languages such as Smalltalk [Gol83], C++ [Laf90], or CLOS [Bob88]. However, the syntax and semantics of POCONS is similar to both CLOS and TELOS [Pad91] (thus, there should be a shortened learning curve for programmers familiar with either systems). POCONS can be used to develop hybrid symbolic/connectionist systems, since it is embedded in Lisp which has been used extensively for symbolic inference. It is also extensible, because it allows a user to interactively create new neurons and rebuild the neural network: a feature not available in Neula or NSL. Also, unlike Neula and NSL, POCONS supports persistence which supports perusal of objects, frees the application programmer from database complexities, and, as will be shown in the next chapter, also makes the retraining of networks unnecessary through the reuse of trained networks. Likewise, unlike

Neula and NSL, POCONS uses objects to represent relationships between different elements of the network. Thus, the relationships as well as the neurons can be examined in the database after the application has terminated.

POCONS is a declarative language in that the programmer simply specifies the structure of the network, enters a command to make the system build the network's internal structure, and initiates execution of a simulation.

6.1.1 The Connectionist Model

The fundamental concept behind connectionism [Fel82] is that individual neurons do not transmit large amounts of symbolic information. Instead, they compute by being appropriately connected to large numbers of similar units. In this way, connectionist systems model the way the brain processes information. It is therefore unlike pattern-directed inference mechanisms which are used to implement production rule [Bro85] and logic-based [Ste86] systems. The inferencing mechanisms in Prolog (logic-based) and OPS-5 (production rule) use pattern-matching to perform operations like unification which assigns facts to variables by matching patterns presented in rules. Connectionist systems use mathematical formulas to propagate values which do the inferencing.

In addition, connectionist simulators such as P3[Rum86], Mirrors/II[D'A88], RCS[Fel88], Neula[Flo90], Slonn [Wan90], NSL [Wei91], and POCONS allow users to execute connectionist simulations and examine in a stepwise manner how connectionism models the reasoning of the brain.

There are many different ways of representing and training neural networks [Was89]. However, recent research in machine learning theory indicates that the learning of higher level knowledge from raw data, as opposed to pattern recognition-based learning, is extremely time consuming and difficult [Pit88]. Recent attempts have been made to combine symbolic with neural techniques. The symbolic component can be used for complicated inference, and the neural component can be used to provide a basic knowledge representation scheme that works for noisy or incomplete data. As mentioned previously, POCONS can support these hybrid systems; however the focus of this chapter will be solely with connectionist applications.

6.1.2 Object-Oriented Connectionist Model

Stroustrup defines object-oriented programming [Str91] as expressing the distinction between general properties (eg. a shape has a color, it can be drawn, etc.) and specific properties (eg. a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.). Languages which allow these distinctions to be expressed and used support object-oriented programming. Other languages don't.

Object-oriented modelling is advantageous for connectionist systems, because connectionist systems are used to model real-world behaviors which have general and specific distinctions. Thus, POCONS is based on the object-oriented connectionist model. In POCONS, the user does not specify any procedural information about the network's execution. The model only requires that the user specify the neurons which represent the components of the network, their attributes, and relationships between them. POCONS can then be instructed to generate a neural network. Queries can be made on the network which initiate connectionist simulations.

The underlying POCONS system translates connectionist objects into sets of neurons that represent the class hierarchy and attributes. Each class has a neuron associated with it, and likewise the class neuron has weighted *is-a* links to the neuron which represents its superclass. Also, a neuron is created for each class slot-value pair (eg. (speed . 60)) which has links to its class and the class has links to it. In section 6.1.3, an extensive example will be presented which shows how to build an object-oriented connectionist network in POCONS.

6.1.3 Building a Connectionist Model

The following sentences were originally presented by Anderson[And76]:

John is a tall lawyer. (6.1)

The lawyer owned a dog. (6.2)

John kicked a model. (6.3)

The model's name is Jane. (6.4)

The model John kicked owns a car. (6.5)

The knowledge in these sentences can be represented in POCONS in the following way:

```
(defdbneuron person (newron)
  ((owned initform nil)
   (height initform nil)))

;NEWRON is the top-level object containing
;audit information used in converting the
;objects to a connectionist network.
(defdbneuron lawyer (person)
  ((owned  initform 'dog)))

(defdbneuron john (lawyer)
  ((kicked initform 'model)
   (height initform 'tall)))

(defdbneuron model (person) ())

(defdbneuron jane (model)
  ((owned  initform 'car)))

(defdbopposites lawyer model)
```

In the above code, *Defdbneuron* is the defining component for the creation of a persistent neuron. It has the following form:

```
(defdbneuron neuron-name (superclasses) (slots))
```

The *neuron-name* will be used as a symbol that identifies the neuron. The *superclasses* specify the class or classes from which the neuron inherits. The *slots* describe the explicit relationships that the neuron will have. Slots are specified as a list containing *slot-names* and initial values. For example, the neuron *Jane* has a slot *owned* which has the value *car.*

Defdbopposites indicates a relationship between two neuron types and is defined as follows:

```
(defdbopposites neuron-name neuron-name)
```

The *neuron-name* arguments must have been defined as neurons using *defdbneuron.* As a result of the use of *defdbopposites*, the system will place a negative link between the two specified neurons in the network. The negative link will keep the attributes of each neuron

from being associated with the other. For example, *(defdbopposites lawyer model)* will keep the *lawyer*'s attribute *tall* from being associated with the *model.*

The use of *defdbopposites* generates a persistent object containing the specified information. Thus when reusing a neural network, one need only to reopen the database and all the information will be loaded on demand by the persistent object system into primary memory.

6.1.4 Conversion of Objects to a Connectionist Network

Once the neuron definitions have been loaded into POCONS, the user can instruct the system to convert the neuron classes to the underlying internal representation by executing the *build-neural-network* function.

The conversion algorithm examines each object and a neuron is created for each neuron name and for each slot attribute and value. The *slot-attribute pair* corresponds to *(character initform '((good . .7) (evil . .3)))* which would be converted into two neuron's: *(character good .7)* and *(character evil .3).* It then creates forward links from each subclass neuron to each superclass neuron. Links are also created from class neurons to their slot neurons. Pseudocode for this procedure is as follows:

```
For each defneuron
    1. Convert neuron name to a neuron and link to superclass neurons
    2. For each slot attribute
       a. convert each slot-attribute pair to a neuron and link it
          to its class neuron
    3. Create back links from subclass neurons to superclass neurons
```

The following formal specification of the algorithm uses the formal high-level algebraic specification technique [Geh86]. In the specification, a *neuron-object* is the instance created by a call to *defdbneuron* as shown earlier which can be translated into many neurons, and a *neuron-struct* is the structure created for a single neuron.

```
TYPE conversion-of-objects-to-neurons

EXTERNAL OPERATIONS make-instance, write-instance-to-database, member, cons
                    superclass-of, set-link, create-structure,
                    return-class-slots
```

```
INTERNAL OPERATIONS defdbneuron, make-neuron-struct,
                    build-neural-network, convert-neuron-obj,
                    get-superclass-obj, link-slot-to-class,
                    link-class-to-superclass

SYNTAX

  defdbneuron: arglist  -> neuron-object
  make-neuron-struct arglist -> neuron-struct
  build-neural-network:  -> neuron-struct-list
  convert-neuron-obj: neuron-obj -> neuron-structs
  get-superclass-obj: neuron-obj -> superclass-obj
  link-slot-to-superclass-obj: slot X class -> nil
  link-class-to-superclass: neuron-struct X  neuron-struct -> nil

SEMANTICS

  VAR sub, super, n: neuron-struct
      k: neuron-object
      slot: neuron-object-slot

  AXIOMS

    defdbneuron(arglist) = neuron-obj-list:= cons(neuron-obj-list,
                           write-instance-to-database(make-instance(arglist)))

    make-neuron-struct(arglist) = create-structure(neuron, arglist)

    get-superclass-obj(neuron-obj) = get-neuron-name(superclass-of(neuron-obj))

    link-neuron-class-to-superclass(sub, super) = set-link(is-a, sub, super)

    convert-neuron-obj(neuron-obj) = k:= get-superclass-obj(neuron-obj)
                                     IF NOT member(name(neuron-obj),
                                                   neuron-struct-names)
                                        THEN convert-neuron-obj(k)
                                        n:= make-neuron-struct(neuron-obj)
                                        link-neuron-class-to-superclass(n, k)
                                        FOR EACH slot IN return-class-slots(n)
                                            link-slot-to-class(slot,n)
```

As the connectionist simulation executes, the inheritance mechanism propagates values from class to subclass (Figure 6-1), due to the training algorithm (discussed in the next section) acting upon the downward links in the neural network which were initialised by the conversion process. The back links from subclass to class have a fraction (0.25) of the weight of the downward links. The back links enable, for example, the activity value for *person*, if

the one for *lawyer* is enabled. However, the links from class to subclass have larger weights to make the model consistent with the inheritance mechanism in object-oriented systems. Figure 6-2 illustrates the internal representation of the simple network defined previously.

Activation Propagation

Activation propagation is the method employed by POCONS to train the network. It is a common training algorithm which is also used at least to some extent in Mirrors/II, RCS, P3, SLONN, and Neula. Activation propagation as used in POCONS is based mostly on the way it is used in Neula. It is based on the *Hebbian* [Heb49] idea of cell assemblies: a Hebb rule acts so as to strengthen often-used pathways in a network, and was used by Hebb to account for some of the phenomena of classical conditioning. Let j refer to any one of the input lines to a neuron, and let i refer to any one of the neurons in the network. A weight associated with a connection is therefore increased whenever the *jth* input line is active and neuron i is firing. There is no facility for decreasing weights.

The Hebbian learning rule is that w_{ij} is strengthened by correlated input to the cell and output from the cell. The change w_{ij} of the connection weight at time t between the *jth* and *ith* cells under Hebbian learning is therefore:

$$W_{ij}(t) = rx_j(t)Out_i(t) \tag{6.6}$$

where x_j is the input line activity from the *jth* cell, Out_i is the output of the *ith* cell, and r is the learning rate; $r = 0.1$ is usually chosen.

Hebb proposed his learning rule for networks of real neurons. The biological basis of memory is still not clear, but the Hebbian learning technique can be exploited in a computer

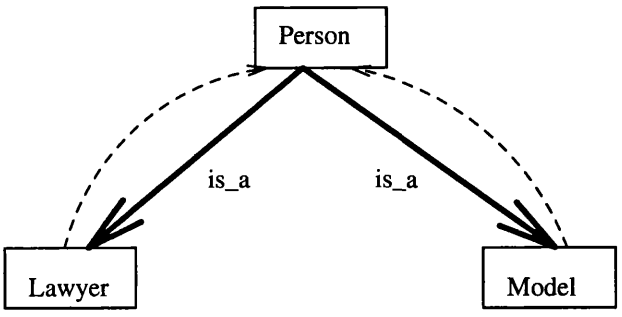


Figure 6-1: Inheritance in POCONS

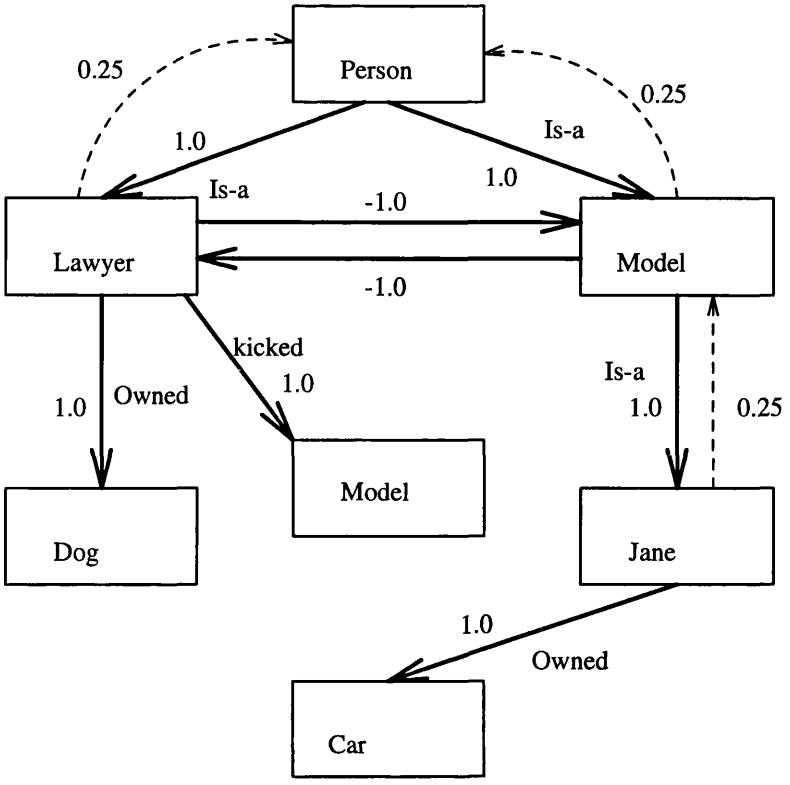


Figure 6-2: Internal Representation of a Connectionist Network

to perform inferential operations as is done in Mirrors/II, RCS, P3, SLONN, Neula, and POCONS.

The variant of Hebbian learning used in POCONS is called activation propagation. The main differences between it and Hebbian learning is that it propagates values instead of weights and those values can be increased and decreased. Activation propagation is used to have values of related nodes affect one another to perform inferences.

In activation propagation each node has an activity value indicating its level of truth. In the object-oriented connectionist model, the activation values get propagated throughout the network according to the rules of inheritance: subclass nodes inherit from their superclasses, and slot nodes inherit from their classes.

Initially, nodes have activity values, by default, of zero. Some nodes get values of 1 or -1 stored in them as a result of the query being made on the network. The user may specify how many iterations they want the activation propagation algorithm to perform. The number of iterations determines how far the user wants the inheritance values to spread throughout the network.

To compute the change in a node's activity level, the following formula is used for each

node:

$$a_i^{k+1} = \delta(a_i^k) + thres(\sum_j w_{ij} a_j^k) \tag{6.7}$$

The a_i^k represents the kth activity level of node a_i. The summation means to sum the multiplication of the outgoing arcweights with the activity levels of the nodes they link to. The δ function computes $1 - abs(a_i^k)$ to guarantee that there is no drastic change in the value when there is a value close to 1 or -1. The *thres* function is a threshold function that uses truncation to keep the activity value boundaries between $[1, -1]$.

Please note that even though there are many different varieties of neural networks, they do have the following commonalities: weighted links, independent neurons, and a simulation algorithm which multiplies the weights against the neuron values. Activation propagation is only one of many techniques available. It was chosen, because it applies well to inheritance in object-oriented systems.

Neural networks also make use of training algorithms such as backpropagation [Rum86] which modifies the weights to make the network recognize different patterns. Successful training of a neural network using backpropagation can require many hours of CPU time. Backpropagation is not included as a utility in POCONS, because there are many different types of training algorithms which developers may use. If a training algorithm other than activation propagation is desired, it can be added by the user by interfacing with the POCONS system. In the next chapter, a chaotic model that was added to POCONS by the author will be presented.

6.1.5 Example Simulation

Once the the connectionist network has been built, simulations can be executed using the *query-net* command which has the following form:

```
(query-net activities-to-set values-to-display cycles)
```

Query-net requires three arguments. The *Activities-to-set* argument is a list of the neurons and initial activity values. If no value exists for a node in the *activities-to-set* list, the value will be set to one. The default value for all activities not listed is zero. *Values-to-display* is a list of the labels associated with the neurons whose values are to be displayed,

because their changing values are of interest to the user during the simulation. *Cycles* indicates the number of iterations the simulation will use the activation propagation algorithm.

During the simulation, values propagate around the network to indicate the truth levels of nodes in the system. In the first query given below, the activity value for *Jane* is set to 1.0, and the simulation is executed for five cycles. Each set of lists corresponds to a cycle in the execution of a simulation. During each cycle, values propagate out from the *(is-a jane 1.000000)* neuron to other neurons that are either linked directly to it or have an indirect connection to it.

The following call to *query-net* initiates a five cycle simulation on the network defined in Section 6.1.3:

```
eulisp:0:act!2> (query-net '((is-a . jane)) nil 5)
(is-a model 0.330000)
(owned car 1.000000)
(is-a jane 1.000000)

(is-a person 0.108900)
(is-a lawyer -0.330000)
(is-a model 0.551100)
(owned car 1.000000)
(is-a jane 1.000000)

(is-a person 0.173917)
(owned dog -0.330000)
(is-a lawyer -0.626274)
(is-a john -0.330000)
(is-a model 0.896259)
(owned car 1.000000)
(is-a jane 1.000000)

(is-a person 0.247517)
(owned dog -0.749604)
(is-a lawyer -0.977629)
(height tall -0.330000)
(kicked model -0.330000)
(is-a john -0.749604)
(is-a model 1.000000)
(owned car 1.000000)
(is-a jane 1.000000)

(is-a person 0.253072)
(owned dog -0.994398)
(is-a lawyer -1.000000)
```

```
(height tall -0.832234)
(kicked model -0.832234)
(is-a john -1.000000)
(is-a model 1.000000)
(owned car 1.000000)
(is-a jane 1.000000)
```

The resulting values for each neuron represents the level of truth that can be inferred about *Jane* and the various relations in the net that get activated. If a neuron's activation value is zero, then it is not displayed.

The next query illustrates how one can selectively ask the net questions and have the output only list certain nodes. The next query asks the following question:

Who did the dog owner kick? (6.8)

```
eulisp:0:act!7> (query-net '((owned . dog)) '(kicked) 5)

(kicked model 0.330000)

(kicked model 0.798489)

(kicked model 1.000000)
```

It takes the simulation three cycles before it has any kind of information pertaining to the query to display. The final value, *(kicked model 1.000000)*, occurs at the fifth cycle. It indicates that the network infers that the *model* was kicked by the *dog owner*. The certainty factor increases after each step, because the links to *(kicked model val)* propagate larger values as the simulation proceeds.

6.2 Implication in a Connectionist Model

Implication allows rules to be specified which help to solve a problem and build knowledge bases. The implication rules described in this section is an extension that I made for this thesis to the modelling capabilities of the activation propagation algorithm. A knowledge base requires a set of facts, a set of rules, and an inference engine which applies the rules to the facts to produce inferential results. The implication mechanism described in this

section, provides both a construct for defining rules and an inference engine mechanism which applies the rules to the facts stored in the neural network.

For the purposes of this thesis, the ability to develop knowledge bases in PSE is important so that simulations can be written that need to consult expert knowledge. The integration of knowledge-based systems and simulation has been shown to be useful by O'Keefe [O'K86] and others. O'Keefe lists three useful applications for this integration:

1. Generating new simulation tools from combining existing simulation and knowledge-based methods.

2. Advice giving systems for inexperienced simulators particularly in areas of experimentation and analysis.

3. Intelligent front-ends to a simulation.

Modsim [Her92a] is a Modula-2 based simulation language which has a component that allows the simulation programmer to interface with a Prolog-like system called ModLog [Whi92a] which, like POCONS, can be used to represent expert knowledge for consultation by the simulation. Like Modsim, PSE integrates knowledge representation with simulation libraries. Unlike Modsim, PSE's knowledge-based representation component uses a connectionist representation as opposed to Modsim's Modlog component which uses a logic-based representation. The advantage of using a connectionist representation is that it is more efficient [Ali93] and is explicitly parallel [Rum86]. Large simulations normally require significant amounts of CPU time, and for this reason the connectionist model was chosen to be included in PSE, because as will be shown, its explicitly parallel nature allows for significant performance improvements when executed on a parallel machine. Experiments using parallelism have been conducted in this thesis using POCONS and will be described in chapter 7.

Originally, POCONS was implemented to represent facts and connections through an object-oriented specification, but recently the ability to represent implication has been added to it. The motivation for representing implication in POCONS stems from experience with langauges such as Prolog[Ste86] which represents implication in logic and OPS-5 [Bro85] which represents implication in production rules. Other connectionist languages allow users to build the information contained in rules, but it must be built in an implicit rather than explicit manner. The construct for implication, *defimplies*, can be used to

represent knowledge explicitly. This section describes the implication construct, its implementation, how the system converts it into the connectionist representation, and presents an example of its usage.

The only other connectionist system which provides constructs for rules is DCPS [Tou86] which does the matching of rules at run time. POCONS resolves the matching at compile time for greater efficiency.

DCPS is based on a working memory model which contains the facts used for inference. It is based on the standard rule-based system approach which has a set of rules, a set of facts, and an inference engine which does the matching of rules to facts at run time. The advantage of matching of predicates at run time is that it allows rules to fire that partially match, allows rules which become satisfied due to dynamic behavior of the system, and allows new rules to be added dynamically. However, due to its dynamic nature, there is a greater amount of run time complexity than there is in the method described in this section which resolves rules at compile time.

6.2.1 Implication

The implication construct in POCONS is a *new* addition to activation propagation that I designed and implemented for this thesis. The construct is called **defimplies**. It allows a set of *conditions* that if active will cause the set of *results* to be activated as well. The syntax for **defimplies** is as follows:

```
(defimplies (condition1 [conditions])
   =>  (result1 [results]))
```

A *condition* contains a condition name followed by a list of variables corresponding to the neurons it depends on to be satisfied. For example, in the condition *(nand-gate input1 input2 output)*, the first element, *nand-gate*, is the name for the condition that will be satisfied if the other conditions and result clauses are satisfied. *Input1*, *input2*, and *output* correspond to variables that must be satisfied based on their use in the result clauses to be made true. Multiple conditions are *anded* together as in Prolog. Each result must consist of variables or constants which will be used to match against values stored in neurons. For example, in the result clause *(transistor input1 X output)*, the values for *input1* and *output* must will be the same as for the condition. Each variable represented in the conditions must

appear in at least one of the result clauses. However, result clauses can contain variables that don't exist in conditions (eg. *X*). Conditions and results are *not* order dependent.

The *defimplies* primitive can only be used after a connectionist network has been built by using *(build-neural-network)*. The system interprets *defimplies* by creating a partition of links between the conditions and the results. This partition is built by the system *between* links created by the definition of the network and links created due to the conditions and result clauses in *defimplies*. Figure 6-3 illustrates the representation of a neuron in this *partitioned* network. The neuron in Figure 6-3 labeled *n* has arrows leaving it to indicate links to other neurons. The arrows leaving the circle inside the neuron represent connections built to the neuron as a result of one or more *defimplies*. The connections in this *rule-partition* will be used to compute the activation value for the neuron, because, out of semantic necessity, they take priority over the other ones.

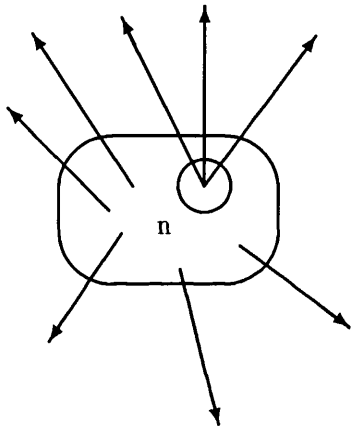


Figure 6-3: Neuron with Partitions

Since the clauses in both the condition and result lists are *anded* together, it is necessary for the system to produce links only for those neurons whose values match the values expressed in the conditions. For example, if a result clause has a variable or constant that is the same as that of another result clause, then only neurons that have values that meet both requirements will be used.

If a neuron is shown to meet the requirements of the result clauses, then links are made from the neurons listed in the condition clauses to that result neuron. This procedure is executed at load time, after a call to *build-neural-network*.

```
For each condition
    match variables in the result class against neuron values
```

```
    if it matches compare it to the other conditions

    if it matches all the conditions then set a link from the
       matching neuron to each one of the neurons in condition clause
end for loop.
```

The following is a high-level formal algebraic specification for the algorithm:

```
TYPE build-implication-links

EXTERNAL OPERATIONS get-all-matches, slots-match, get-neuron-value, cons,
                    set-links

INTERNAL OPERATIONS defimplies, set-rule-links,
                    match-all-results, match-slots,
                    vals-match-results

SYNTAX

  defimplies: conditions X results -> NIL
  match-all-results: conditions X results -> NIL
  match-slots: condition X neuron-list -> neuron-list
  set-rule-links: neuron-struct X neuron-struct -> NIL
  vals-match-results: slot X neuron-struct -> nil

SEMANTICS

  AXIOMS

   defimplies(conditions, results) =
              match-all-results(conditions, results)

   match-all-results(conditions, results) =
    neuron-list := get-all-matches(results)
                FOR EACH result IN results
                  neuron-list := match-slots(result, neuron-list)
                ENDFOR
                cond-list := get-all-matches(conditions)
                FOR EACH condition IN conditions
                  cond-list := match-slots(condition, cond-list)
                ENDFOR
                set-rule-links(cond-list, neuron-list)

   match-slots(slt, neuron-list) =
                 newlis:= NIL
                 FOR EACH neuron in neuron-list
```

```
                IF vals-match-result(slt, neuron)
                   THEN newlis:= cons(neuron, newlis)
               RETURN(newlis)

vals-match-result(slt, neuron) =
               FOR EACH slot in get-neuron-slots(slt)
                   IF slots-match(slot, neuron)
                      THEN RETURN(TRUE)

set-rule-links(conditions, neuron-list) =
             FOR EACH neuron in conditions
                 FOR EACH n in neuron-list
                     set-links(conditions, neuron, n)
```

When the activation propagation [Bur92] training algorithm is then executed, the system will look first to see if there are links associated with rules in the current neuron. If rule links exist, it uses them to calculate the neuron's activity value. Otherwise, it uses the links associated with the original network configuration. Thus, the components of a neuron are represented internally as follows:

```
(defstruct neuron-struct ()

   ;symbolic arc label name (eg. is_a)
  ((nlabel initform nil initarg nlabel accessor nlabel)

   ;symbolic value associated with it (eg. person)
   (nval initform nil initarg nval accessor nval)

   ;integer id for the neuron
   (node-num initform nil initarg node-num accessor node-num)

   ;The neurons truth level
   (activity-level initform 0 initarg activity-level accessor activity-level)

   ;The list of connections built from rules
   (rule-list initform nil initarg rule-list accessor rule-list)

   ;The downward inheritance links
   (connect-list initform nil initarg connect-list accessor connect-list)

   ;The upward inheritance and semantic links.
   (back-pointers initform nil initarg back-pointers accessor back-pointers))

   constructor make-neuron-struct)
```

The only difference between evaluation of rules at load time versus evaluation at execution time is the time at which evaluation occurs. A run-time rule evaluator, as in DCPS, would fire when a query resulted in the conditions of a rule to be satisfied. Whereas in POCONS, the rule links built at load time determine whether a rule is satisfied. The end result is the same.

The reason for having the rule links in the partition of a node have precedence over the node's other links can be justified as follows: a rule is made to answer specific questions about the nature of the facts in the knowledge base. Therefore, if a rule is satisfied, it should take precedence over the other relationships specified between facts in the knowledge base.

6.2.2 Implication Example

The following example is based on a Prolog example due to Sterling and Shapiro [Ste86]. It consists of a database of facts on components of logic circuits as illustrated in Figure 6-4. It also has rules that when applied to the facts will determine which components make up the inverters, and-gates, and nand-gates in the circuit.

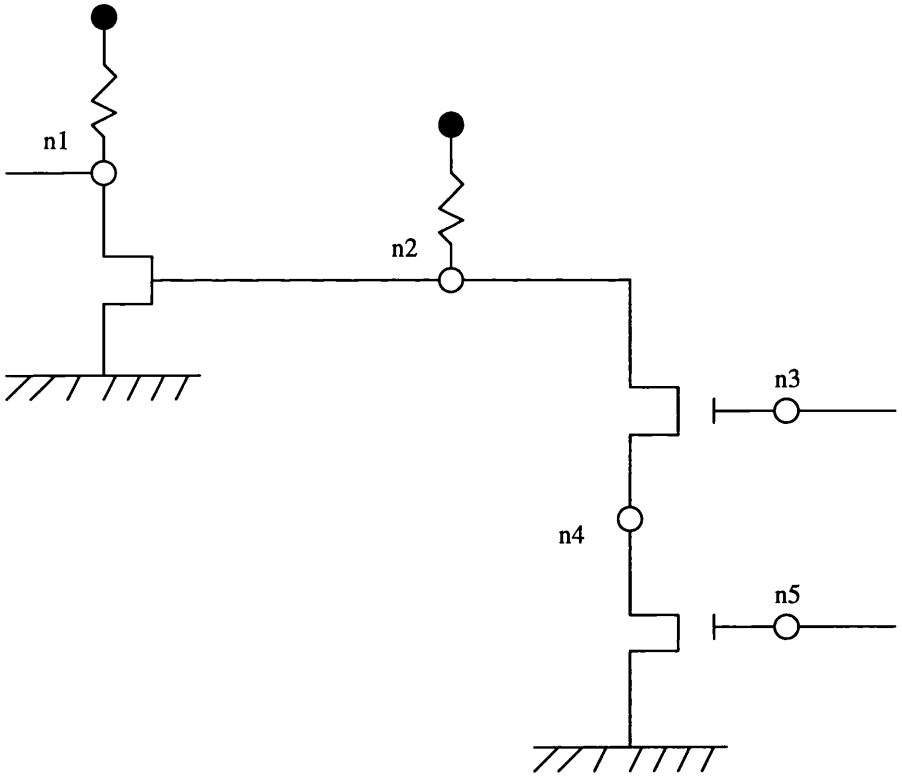


Figure 6-4: logic circuit

The code segment below shows how to define the classes for *resistor* and *transistor* in

POCONS. *r1* and *t1* are definitions for instances of *resistor* and *transistor* respectively. *End1* and *end2* are components which make up the two parts of a resistor. In the case of *r1*, *power* is the *end1* part and *n1* is the *end2* part. In like manner, *n2*, *ground*, and *drain* correspond to *gate*, *source*, and *drain* in *t1* respectively. In the interest of brevity, the code for all of the facts and rules is not listed in full, however after the execution of *(build-neural-network)* the network of neurons will be built.

```
(defneuron resistor (newron)
  ((end1 initform nil)
   (end2 initform nil)))

(defneuron r1 (resistor)
  ((end1 initform 'power)
   (end2 initform 'n1)))

(defneuron transistor (newron)
  ((gate initform nil)
   (source initform nil)
   (drain initform nil)))

(defneuron t1 (transistor)
  ((gate initform 'n2)
   (source initform 'ground)
   (drain initform 'n1)))

(build-neural-network)
```

After the network is built, the *defimplies* for *nand-gate* can be issued as in the following code segment:

```
(defimplies ((nand-gate input1 input2 ouput))
  => ((transistor input1 X output)
      (transistor input2 ground X)
      (resistor power output)))
```

As a result, links will be attached from the *nand-gate* neuron linking to the neurons associated with *transistor* and *resistor* instances which satisfy the variables and constants listed in the result field of the *defimplies*. Note that the *X* variable only needs to match elements from the two different *transistor* neurons. It isn't used in the condition.

After the *defimplies* has been executed, a call to *query-net* will cause the activation

propagation algorithm to make use of the connections between neurons to propagate the implied knowledge throughout the network.

The following listing is produced after three iterations of the activation propagation algorithm over the circuit-based network.

```
(end2 n1 -1.000000)
(is-a r1 -1.000000)
(end2 n2 1.000000)
(is-a r2 1.000000)
(is-a transistor 1.000000)
(drain n1 -1.000000)
(gate n2 -1.000000)
(is-a t1 -1.000000)
(drain n2 1.000000)
(source n4 1.000000)
(is-a resistor 1.000000)
(gate n3 1.000000)
(is-a t2 1.000000)
(drain n4 1.000000)
(gate n5 1.000000)
(is-a t3 1.000000)
(is-a nand-gate 1.000000)
```

Components in the listing that have positive values are ones inferred by the system to be parts of the nand gate. Components with negative values are those which do not meet the requirements of the implication rules to compose a nand gate. This example has shown how the facts defined as objects and the implication rules can be used to do Prolog type programming. One big advantage that neural systems have over Prolog is that their training algorithms are explicitly parallel [Rum86] and can therefore be applied to parallel processing hardware for execution with little difficulty as will be shown in the next chapter.

6.2.3 Use Of Neuron Rules in a Simulation

With the implication primitives implemented, a PSE simulation can now exploit these inferencing capabilities. In many simulations, expert knowledge is needed to make a decision about which of several choices the simulation should choose. For example, in a military simulation, if a tank is under fire and it reaches a fork in the road, the knowledge in the rules can be used to determine which path to take. If the tank is evading attack, then certainly it will want to take the road which is going away from its attackers. It will also

want to choose a road which is more concealed. Knowledge like this can be represented in rules and used to make decisions about which path to take. The following pseudocode shows how it can be done:

```
Consult query on roads to take in evasion given the
        current location
Choose the road that has the most positive attributes that
      support knowledge about evasion
```

The values of the attributes can be dependent on the state of the simulation. No work has been done in this thesis in developing such a simulation. Nevertheless the utilities for discrete-event simulation and connectionism have been tried and tested. Also, both components can be loaded into a EuLisp image so that they can be used together. The development of a meaningful simulation that combines a knowledge base and discrete-event simulation is a large project and is beyond the scope of this thesis; however, a small example will be presented now just to illustrate the interface. The following code shows how an event written for the event-based utilities from chapter 4 can access the tiny knowledge base from section 6.1.3.

```
;;; Predicate using the kb.
(defun is-abusive (name)
   (reset-neural-network)
   (query-net (list (cons 'is-a name)) '(f) 5)
   (if (= (activity-level (get 'kicked 'model)) 1.0)
       t
       nil))

(defmethod review-background ((b boss) (pe potential-employee))
  (if (is-abusive (name pe))
      (do-at (current-time) `(dump-resume-into-trash ',b ',pe))
      (do-after 1000.0 `(schedule-interview ',b ',pe))))
```

The predicate *is-abusive* is given the name of the person to do a check on, and since the only information related to abusive behavior in the knowledge base is *(kicked model xxx)*, does a check on that neuron to see if the system can infer whether such an action took

place. If it has taken place, the event *review-background* schedules the event *dump-resume-into-trash* to occur immediately. If there is no evidence of abuse, the system schedules the event *schedule-interview* for sometime in the far distant future.

Below is an example of how a knowledge base can be reset and how the accessing of an activity value works in the POCONS system. Resetting is important, because it allows the user to query the network with new settings without rebuilding the entire network.

```
eulisp:0:pse-net!4> (print-vals *neuron-list*)
(is-a db-converse 0.000000)
(is-a db-negconverse 0.000000)
(is-a person 0.270958)
(owned dog 0.000000)
(is-a lawyer 0.108900)
(height tall 0.000000)
(kicked model 0.000000)
(is-a john 0.000000)
(is-a model 0.748122)
(owned car 1.000000)
(is-a jane 1.000000)
eulisp:0:pse-net!4< (() () () () () () () () () () ())

eulisp:0:pse-net!5> (reset-neural-network)
eulisp:0:pse-net!5< (0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000)

eulisp:0:pse-net!6> (query-net '((is-a . john)) '(a) 5)
eulisp:0:pse-net!6< ()

eulisp:0:pse-net!7> (activity-level (get 'kicked 'model))
eulisp:0:pse-net!7< 1.000000
```

6.3 Summary

POCONS is a language that supports the development of connectionist networks using persistent objects. It was added to PSE to provide support for the development of knowledge bases to be consulted by a simulation. Like other connectionist simulators, it uses a form of activation propagation. However, it has several advantages over those other systems:

1. persistent objects
2. interactively extensible
3. relationships are objects

The POCONS system creates persistent objects to represent all neurons and relationships in the neural network. These objects can then be examined in the database and reused (as will be shown in the next chapter).

Another advantage is that POCONS is Lisp-based, so it supports symbolic inference mechanisms.

POCONS is limited in that it has no built-in learning mechanisms. Also, it contains both a prototype persistent object system (PSE) and a preliminary implementation of EuLisp[Con] as underlying layers. Thus, it is not a good performer for large-scale applications. However, it does serve its purpose as a prototype system which shows the effectiveness and utility of its design as a persistent object-based connectionist simulator.

POCONS was added to the EuLisp version of PSE to extend its modelling capabilities. Simulation and knowledge-based systems can be combined to allow simulations to exploit expert knowledge. POCONS is an object-based connectionist language which provides primitives for developing knowledge bases which can be consulted by a simulation developed using discrete-event, process, or Petri net utilites in PSE (see chapters 4, 5, and 8). The *defimplies* primitive shows that rules can be implemented in a connectionist model without symbolic pattern matching. The main advantage of using the connectionist model over a symbolic one is its explicitly parallel nature.

The object-oriented connectionist model and its realization through POCONS primitives has been described. The technique for implication through partitioned connections as implemented in POCONS has been described. Neurons that match the rules are determined statically and connections between neurons as a result of the rules are stored in partitions separate from the links built between factual information. When using activation

propagation to train the network, the system gives priority to connections associated with rules.

Two examples have been presented. The first one was simple, but was useful as an aid to describing the system. The second was a logic circuit simulation which though it is not an actual real-world application, it is realistic in the nature that it is large and it is similar to the kinds of models that get used in practice.

The next chapter will cover extensions made to the connectionist component of PSE to support parallelism, reuse (a particularly good use of persistence), and chaos-based models which extends POCONS's inferencing mechanism.

Chapter 7

Extended Connectionist Simulation

This chapter is a direct followup to chapter 6 in that it describes extensions made, for this thesis, to the POCONS connectionist simulation package which was added to PSE to provide support for knowledge bases. Once again, the goal of this thesis is to merge parallelism and persistence in support of several and hybrid simulation paradigms. The advantages of this goal as related to the connectionist model will be shown in this chapter through the improvement of performance through parallelism and the reuse of trained neural networks through persistence. The extensions were carried out to support a different kind of modelling (chaos-based), parallel simulation (on SIMD and MIMD machines), and storage and reuse of neural network objects which relies heavily on persistent objects.

The chaos-based modelling component was designed to support the modelling of systems which exhibit chaotic behavior such as financial markets, weather patterns, and ocean waves. These and other models have been shown by researchers in their various fields to exhibit chaotic or persistently unstable behavior [And88].

Another extension, which will be discussed first, is the support for parallel simulation of neural networks on SIMD and MIMD architectures through compilation techniques that convert the neural network into code that uses parallel constructs.

An altogether different extension was the addition of a feature that supports storage and reuse of persistent neural networks. As will be described in the next section, one of the major advantages of this feature is that it supports the reuse at a later date of trained neural networks which makes retraining unnecessary and therefore improves performance.

7.1 Language Supported Storage and Reuse of Persistent Neural Network Objects

The new feature (which is only available in POCONS) described in this section allows the state of the neural network to be checkpointed, stored as persistent objects, and reused from its checkpointed state. This feature provides an elegant facility for storing data from trained neural networks and reusing them in later executions. Since trained neural networks can take many CPU hours to create, it is useful to have an elegant manner to store and reuse these objects.

7.1.1 Checkpointing and Storage

While executing a neural network simulation, it can be desirable to save the state of the network at various intermediate stages to examine its evolutionary development. While the training algorithm executes, the state of the neurons and/or weights get altered. The checkpointing feature saves the state of the neurons to a persistent object store but allows the simulation of the network to continue execution. Objects in the persistent object store can later be examined through queries to an object-oriented database system.

Likewise, when execution of the neural network has ceased, the state of the network can be saved using the persistent object checkpointing system. Then the network can be loaded and reused at a later date in the same state it was in when it was stored saving possibly hours of CPU time. Certainly, a trained network could be stored in files manually, then loaded and converted without the use of persistent objects. However, POCONS's built-in facility for persistent neural network objects, *transparently* stores and retrieves them. Storage and retrieval is carried out by the underlying system on a demand driven basis. It therefore requires little extra application code (opening and closing of databases) to save and load objects as will be shown in a later example.

Execution of the newly built-in function **convert-neurons-to-pos** will cause the checkpointing mechanism to convert the neuron structures into persistent objects by storing neuron data items in persistent slots and converting hard pointers to other neurons to soft pointers that can then be stored in the database as integers and dereferenced through a table lookup. The system creates one persistent object for each neuron and stores all data associated with that neuron in it. Note that these checkpointing objects are of a different class than the ones used to describe the structure of the network as shown in the calls

to **defdbneuron** (in section 6.1.3) where one persistent object is used to represent the information described in the call to it. As described previously, this information is then used to create a neural network for which there will be several neurons used to represent the information in a single call to **defdbneuron**. Likewise, when the neural network then gets converted into persistent objects for storage, there will be a one-to-one correspondence between neurons and persistent objects created by the system to store the information in those neurons.

7.1.2 Reuse

POCONS supports the loading of the state of a previously trained or simulated network into primary memory, and the conversion of it into POCONS's internal neural network representation, so that it can then be reused. Once reloaded and converted, training can be resumed from where it left off. In the conversion process, data items (activity values and semantic information) from persistent objects are stored in the representative neurons. Soft pointers stored in the persistent objects are then converted into hard links and stored in the respective neurons.

After opening the correct databases, execution of the newly built-in function **convert-pos-to-neurons** will convert the persistent objects representing the previously checkpointed neural network into the internal neural network representation so it can be executed. The structure of the executable neural network does not use persistent objects, so it can contain hard pointers. Also, the attributes of the neuron get updated frequently as it executes and there would be too much overhead if the neuron slots were persistent. Thus, by avoiding the storage of frequently updated slots at execution time, performance is greatly improved.

Figure 7-1 represents a flip-flop used by a circuit analysis program written in POCONS which analyzes the basic units of the circuit (eg. transistor, resistor) and determines based on the configuration, which groups of these units make up different types of gates (eg. nand, and, nor, etc.).

Besides improving runtime efficiency, another advantage of resolving rules at load time rather than at execution time is that the rule links can be stored in the persistent object, and reloaded for future program execution. As a result, the rules don't need to be resolved when they have been stored in persistent objects. This ability to store rule links greatly

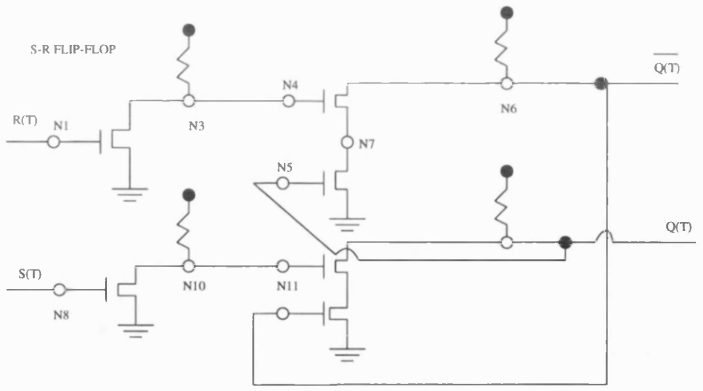


Figure 7-1: Flip-flop used for circuit analysis

improves the performance of the reuse of a neural network as will be shown in the upcoming example.

The output below was generated from a session using POCONS to load and create the neural network code from section 6.1.3. After the network is created, the function **query-net** is used to generate three iterations of the activation propagation algorithm. The network is then converted to persistent objects and EuLisp is exited. Then EuLisp is started up again, and POCONS is loaded. After opening the database, the persistent objects are converted into a neural network which has the same state as it did when it was stored as shown. Further activation propagation iterations are then executed on the network.

```
brad:1 % nufeel
Initialised with: 400000 [128 pages]
stack: 0x1001a2f8 Lim: 0x1001e2f8
Bytecodes compiled on: Thu May 28 14:01:59 BST 1992
EuLisp FEEL: Version (7.04 SystemV) Thu May 28 14:02:36 BST 1992

                    Version Message

     "Bugs fixed, goes faster, version message changed."

         Support the campaign for informative messages

Loading module 'initcode'
Loaded 'initcode'
EuLisp:0:root!0> (!> pse-net)
Loading module 'pse-net'

[ LOADING OF PSE MODULES ]
```

```
Loaded 'pse-net'
eulisp:0:pse-net!0< pse-net

;;;OPENING OF DATABASE

eulisp:0:pse-net!1> (open-dbclass-cache "../neuron/cl.dir")
eulisp:0:pse-net!1< ()
eulisp:0:pse-net!2> (open-classes "../neuron/pmppos")
eulisp:0:pse-net!2< #<stream: 268483904 'r'>
eulisp:0:pse-net!3> (open-objects "../neuron/objects.pos")
eulisp:0:pse-net!3< #<stream: 268483920 'r'>
eulisp:0:pse-net!4> (open-object-cache "../neuron/obj.dir")
eulisp:0:pse-net!4< ()

;;;LOADING OF DEFDBNEURON DESCRIPTIONS

eulisp:0:pse-net!9> (include-forms "../neuron/act-np.em")
including '../neuron/act-np.em'
eulisp:0:pse-net!9< ()

;;; CONVERSION OF DEFDBNEURON DESCRIPTIONS INTO NEURAL NETWORK

eulisp:0:pse-net!10>  (build-neural-network)
eulisp:0:pse-net!10< ()

;;;EXECUTION OF 3 ITERATIONS OF ACTIVATION PROPAGATION.
;;;PRINT-OUTS SHOW STATUS OF NETWORK FOR NEURONS <> 0.0

eulisp:0:pse-net!11> (query-net '((is-a . jane)) nil 3)
(is-a model 0.330000)
(owned car 1.000000)
(is-a jane 1.000000)

(is-a person 0.108900)
(is-a model 0.551100)
(owned car 1.000000)
(is-a jane 1.000000)

(is-a person 0.270958)
(is-a lawyer 0.108900)
(is-a model 0.748122)
(owned car 1.000000)
(is-a jane 1.000000)
eulisp:0:pse-net!11< 4

;;;CONVERSION OF NEURONS TO PERSISTENT OBJECTS FOR STORAGE.

eulisp:0:pse-net!12> (convert-neurons-to-pos)
eulisp:0:pse-net!12< ()

;;;CLOSING OF THE DATABASE.

eulisp:0:pse-net!13> (close-system)
eulisp:0:pse-net!13< ()
```

```
eulisp-handler:0:pse-net!15> (exit)
Exiting EuLisp

;;;NOW A NEW EULISP IS STARTED UP TO REUSE THE SAME NETWORK.

nufeel
Initialised with: 400000 [128 pages]
stack: 0x1001a2f8 Lim: 0x1001e2f8
Bytecodes compiled on: Thu May 28 14:01:59 BST 1992
EuLISP FEEL: Version (7.04 SystemV) Thu May 28 14:02:36 BST 1992

                    Version Message

     "Bugs fixed, goes faster, version message changed."

         Support the campaign for informative messages

;;;LOAD POCONS AGAIN.

Loading module 'initcode'
Loaded 'initcode'
eulisp:0:root!0> (!> pse-net)
Loading module 'pse-net'
Loading module 'standard0'

;;; DECLARE NAMES OF PERSISTENT CLASSES AND OPEN DATABASE

eulisp:0:pse-net!1> (persistent-classes
                       (person lawyer john model jane nnode db-opposites
                        db-converse db-negconverse) "../neuron/cl.dir")

eulisp:0:pse-net!3> (open-classes "../neuron/pmppos")
eulisp:0:pse-net!3< #<stream: 268483904 'r'>
eulisp:0:pse-net!4> (open-object-cache "../neuron/obj.dir")
Reading class db-opposites from the database
eulisp:0:pse-net!5> (open-objects "../neuron/objects.pos")
eulisp:0:pse-net!5< #<stream: 268483936 'r'>

;;;CONVERT THE PERSISTENT OBJECTS TO A NEURAL NET.

eulisp:0:pse-net!7> (convert-pos-to-neurons)
eulisp:0:pse-net!7< ()

;;;PRINT THE STATE OF THE NETWORK.  NOTICE IT IS THE SAME
;;;STATE AS IT WAS SAVED.

eulisp:0:pse-net!8>  (print-vals *neuron-list*)
(is-a person 0.270958)
(is-a lawyer 0.108900)
(is-a model 0.748122)
(owned car 1.000000)
(is-a jane 1.000000)
eulisp:0:pse-net!8< ()

;;;COMPUTE A FURTHER ITERATION OF THE ACTIVATION PROPAGATION
```

```
;;;ALGORITHM.

eulisp:0:pse-net!9> (compute-activation-aux *neuron-list*)
(is-a person 0.477144)
(owned dog 0.108900)
(is-a lawyer 0.350351)
(is-a john 0.108900)
(is-a model 0.899490)
(owned car 1.000000)
(is-a jane 1.000000)
eulisp:0:pse-net!9< ()
```

The above example was kept small for brevity, but it fully illustrates the small amount of application code required for the storage and reuse of neural networks in POCONS. The larger circuit analysis application works in a similar fashion.

7.1.3 Performance Improvements

On a Stardent Titan executing EuLisp [Pad91] it takes 238.94 seconds to load 437 neuron objects, build the neural network, and resolve three rules from the circuit analysis program described in section 6.2.2. It then takes 37.88 CPU seconds to convert the neural network to persistent objects converting the hard links to soft links and storing them and the other neuron values in newly created persistent objects (which the persistent object system transparently stores to secondary memory). In comparison, to reload and build the same network takes 102.42 CPU seconds. Thus, by reusing the objects you get more than double speedup on the amount of time required to initialize the network.

It is interesting to break down the time required to build the network. As a result doing several timings with PSE on different data sets, I found that it takes on average 60 milliseconds to load and instantiate a persistent object using PSE. Thus, since there are 437 objects in the database, it takes approximately 26.22 seconds to load and instantiate them. The remaining 76.2 seconds is required to convert the persistent objects to the internal POCONS representation. Please note that the speedup was attained, because the rules didn't have to be resolved on reloading.

In the case of a backpropagation trained network, more speedup would definitely be attained, because the time to save and reuse the network remains constant, but the time to execute a backpropagation training algorithm requires much greater time than creating a new network from a POCONS language description.

7.2 Parallel Execution on SIMD and MIMD machines

The activation values for the neurons have been implemented to execute in parallel on both a SIMD and MIMD machine architecture. The POCONS system utilizes the Plurals construct for the execution on a SIMD machine and the Futures construct for execution on a MIMD machine.

7.2.1 SIMD Environment

The mechanism used for achieving parallelism on a SIMD machine is called Plurals [Mer92] which is an intermediate between the Paralation model [Sab88] and *LISP [Thi88]. Plurals have been implemented by Simon Merrall on our local dialect of EuLisp [Con] executing on our 1024 node MasPar MP-1011.

MasPar MP-1 Architecture

The MasPar MP-1 [Bla90] is a data-parallel architecture having the following unique characteristics: scalable in terms of the number of processing elements, system memory, and system communication bandwidth; and a "RISC-like" instruction set design that leverages optimizing compiler technology.

The MasPar system contains subsystems for array and independent program execution control, a processor element array, communication mechanisms, a UNIX susbsystem which provides UNIX services, and an channel typed I/O subsystem which allows overlapped I/O messages.

Plurals

Plurals are implemented on the MasPar using the above mentioned synchronous programming model. Plurals are a separate class in Eulisp which are similar in appearance to vectors, but each element is allocated on a separate processor. Two plurals allocated on the same set of processors are said to be *conformant.* Conformant plurals can be operated on in parallel, for example a pair of conformant plurals of integers could be added in parallel and this will create a new plural conformant to the arguments. The function **make-plural** can be used to create a new conformant set or a new plural within a conformant set. Given a positive integral argument, a set of processors will be selected for the new conformant

set and a new plural is allocated on that set of processors. The function **bang** projects a singular lisp object into a plural variable.

```
(setq a (make-plural 10))

#P(() () () () () () () () () ())

(setq c (bang 55.5 a))

#P(55.5 55.5 55.5 55.5 55.5 55.5 55.5 55.5 55.5 55.5)
```

There are parallel versions of serial functions like **car**, **cons**, **nullp**, and **zerop** having the suffix **-s** which act on the individual elements of the plural.

The function **list-to-plural** converts a list to a Plural with the elements of each sublist stored in each plural element. In the following example, the **if-s** function acts on each plural element testing if it is **null** and if not summing up the values. Otherwise it places a zero in the plural slot. In the following code segment, **sum-list-s** computes a vector addition over the plural **list-s**.

```
(defun sum-list-s (list-s)
   (if-s (nullp-s list-s) (bang 0 list-s)
         (+ (car-s list-s) (sum-list-s (cdr-s list-s)))))

(sum-list-s (list-to-plural '((3 2 1) (2 1) (1) ())))
=> #P(6 3 1 0)
```

Also, there is a function called *match* which creates maps which allow data to be moved between plurals of conformant sets. The map describes which elements of the source plural are sent to the destination plural.

Compilation techniques for SIMD Plurals

To execute the simulation of a neural network using Plurals, the system first builds the connectionist network from the object-based representation [Bur92]. The resulting connectionist network consists of a set of structures each one corresponding to a node in the network as described in section 1 of chapter 6.

Once the set of neurons exists, the mapping to plurals begins by first assigning node identifiers to each neuron, and then the network is converted into a pair of lists. Each

list has an entry for each neuron in node id order. The first element of each pair is a list of the ids of the nodes that this node has back links to, and the second element is a list with the corresponding weights for each one of the back links. The list is then converted into a list of plurals containing weights and nodes as specified. Each time an activation propagation is computed for the network, the plurals use a map that represents the activity values for neurons and the weights on their associated links. The activity value is computed for each node in parallel by having the system look at the connections and compute the values. The single operation is to compute the activity value using the map for each node, and the multiple data is the different state and links associated with each node. Thus, to summarize:

- Convert object description into connectionist network.
- Create a list containing lists of all the activity values and weights in the network ((a1 a2 a3 ...) (w1 w2 w3 ...)).
- List is converted to a plural.
- A plurals *map* is made which associates each activity value with the weights on its incoming links.
- The multiply of the activity value times each link's weight is done in parallel.

Figure 7-2 illustrates the plurals map that gets created for the connectionist network for execution on the Maspar. It illustrates the activity values of the neurons in a network and the mapping that exists between them and the weights on their outgoing links. Since part of the activation propagation algorithm is to multiply the value of the weights by the activity values on the outgoing links, this mapping allows the vector multiply operation to multiply the weights by the activity values in parallel.

It should be noted that a full implementation of the paralation model was not available, and as a result, the SIMD operations were limited to those supplied by the plurals system. If a full paralation model had been supported, a complex operator could have been constructed which would compute the full activation propagation algorithm for a single node. This operator could then, under the paralation model, operate in parallel over all the nodes in the network. The result would have been far more parallelism than was produced using the more limited plurals model.

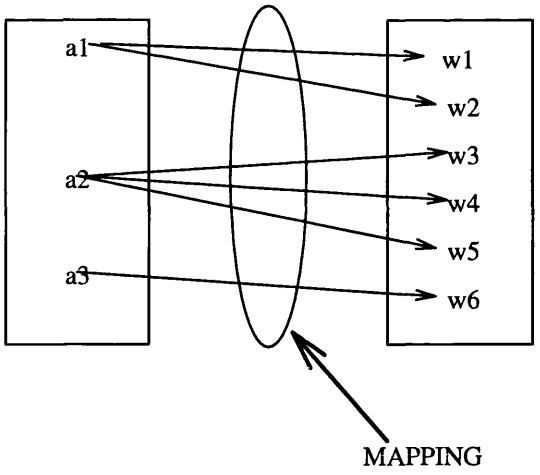


Figure 7-2: Internal Representation for execution of POCONS network using plurals

An evaluation of the usefulness of SIMD connectionist simulation will be made in sections 7.2.4, 7.2.5, and 7.2.6.

7.2.2 MIMD Environment

Our MIMD environment consists of two Stardent Titan multiprocessors each containing four MIPS processors and 32 megabytes of memory.

Stardent Architecture

The Stardent Titan system can configure up to four identical processor boards. It supports three types of parallel processing:

1. Multiprocessing – the parallel execution of multiple processes on multiple processors.
2. Microtasking–the division of a process into multiple threads (microtasks) that can be executed in parallel on multiple processors.
3. The parallel execution of integer operations (address calculations, bitwise operations, etc.) and floating point operations within each Titan server processor.

Futures

Futures [Hal85] are the mechanism used to achieve parallelism for the connectionist system when executed on the Stardent. Futures act as a promissory note allowing a process to request a computation and then continue on its way knowing that the system will provide

it with the requisite information when needed. In the event the Future has not been computed when the process attempts to access its value, the process blocks until the Future has completed execution. Futures in EuLisp are implemented using threads. Threads in EuLisp are implemented in the same manner as in CLIP [Jac90]. Threads are essentially lightweight processes that execute on shared memory in a non-preemtive manner. Thread states can either be *blocked* or in *execution.* The next subsection describes how the system generates Futures that execute the network in parallel on the Stardent.

7.2.3 Compilation technique for MIMD Futures

Since our Stardent system has four processors, the MIMD compilation package partitions the neural network into both three and four sets of equal size to determine which partition produces the best results (see Figure 7-3). It partitions the neurons randomly, because under the activation propagation algorithm, there are no dependencies between neurons. It then places each partition in a Future that will execute the activation values computed in each partition concurrently with the neurons in the other partitions. Specifically, it takes the list of neurons that comprise the neural network and divides it into both three and four lists on separate executions. It then calls a function which maps through the list of neurons passed to it and computes the new activity value for each neuron in the list. The mapping function is called three (or four) times each time within a different Future, so that the three (or four) calls execute concurrently. Once all three Futures have completed execution, the system sequentially updates the activity values for each neuron with its new computed value. The process is repeated for each cycle except that the partition is saved, so that subsequent steps don't have to repartition the network.

There is no need for mutual exclusion amongst the three concurrently executing partitions, because each neuron can have its value computed independently. The number of Futures is limited to four at the most (though it can easily be extended to include more), because any more Futures than the number of processors executing at any given time results in greater overhead and a decrease in performance. This technique has been implemented and tested with a small and a large network. It appears that there are no benefits from partitioning the network in any specific manner, and there is no reason to believe that there would be, because due to the total independence each neuron has in computing its activity value, it is irrelevant as to how the neurons are grouped.

7.2.4 Comparison of architectures

Clearly, due to the large amount of processors, the MasPar architecture has far greater potential for speedup. However, the port of EuLisp to the MasPar is still preliminary. Currently there is no garbage collection, and the EuLisp front end sends the plural operations from a DEC Vaxstation to the MasPar. Thus, the Vaxstation is a major bottleneck for the SIMD computations. However, currently the garbage collector and EuLisp front end are being implemented on the MasPar and upon completion, there should be improved performance.

The Stardent does allow for some concurrency, but with only four processors, the greatest possible speedup is not overwhelming. However, it does provide a vehicle to test the networks on a MIMD architecture and possibly speculate about improvements given more processors.

7.2.5 Results

A 440 node connectionist network which does an analysis of the logic circuits for two flip-flops has been executed on the Stardent using Futures (as described above). Figure 7-3 shows the performance improvements made using the same net partitioned into three and four sets compared to the execution time of the sequential execution of the non-partitioned net. Each point on the graph represents the average over five executions of a 10 cycle query on each configuration. It is interesting to note that the three Future (or partition into thirds) configuration provided better results than the four Future partition. The reason for the this result is that even using three partitions, there is still a main thread which fires off the Futures. In the four partition version, even with four processors, one thread will always be blocked. Whereas with the three partition configuration on four processors, no threads will be blocked. Thus, all threads executed concurrently. The results in Figure 7-3 show *slightly better than double speedup (2.1 to be exact)* in the best case (partition by thirds on four processors). Also note that on a single processor, the partitioned sets were slower (less than 1) than the sequential version. This result can be explained by the extra overhead needed to utilize Futures.

While it may *seem* that the results from a single application are meaningless, since all the activity values are calculated independently on the MIMD machine, the only difference between different applications is the number of neurons. The amount of parallelism remains

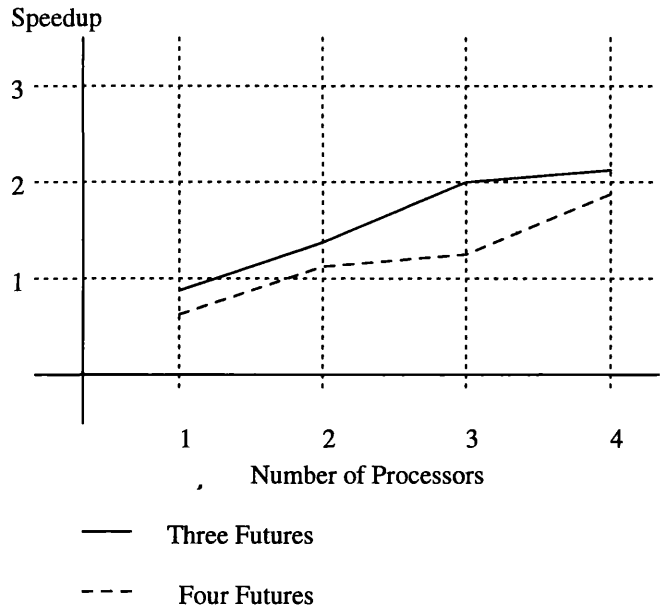


Figure 7-3: Speedup From Connectionist Executions on MIMD Architecture

constant across applications. Therefore, each application will have as much speedup as any other as long as memory usage doesn't get to the point where it causes the virtual memory system to thrash.

A connectionist simulation has also been implemented for the MasPar, but there is no comparable single processor machine to compare it to. It does however, execute faster than the same simulation executed serially on the front-end machine. The problem with SIMD plurals in general, is that SIMD parallelism occurs over the number of connections between nodes. However, if a full paralation model had been available for use, an iteration of the activation propagation could have been made into a vector operator which then could be executed in parallel over the entire network. Using plurals, the connectivity in the applications tested is often only two which limits the speedup regardless of the number of processors utilized. A quick scan through the literature found that there were few large connectionist models that had a high connectivity which is bad news for SIMD plurals enthusiasts. It appears that paralations would be much more useful and productive.

7.2.6 Limitations

As stated earlier, due to the static nature of the rules in POCONS, rules cannot be added dynamically. Also, POCONS will not allow rules to activate neurons if there is a partial match which does not prove to be a problem for the applications written in it to date. Com-

parisons have been made to Prolog, but there is no facility for Prolog-like grammar rules in POCONS. Also, the implementation of EuLisp which POCONS is written in is a prototype as is PSE. Thus, it is not an extremely stable system for production code, but rather it is a system that can be used to experiment with language design and implementation techniques. Nonetheless, the results show that the use of parallelism with the POCONS connectionist model results in significant speedup.

Next, there will be a discussion on a modification to the activation propagation algorithm which produces chaotic neural networks.

7.3 Using Chaos in Neural Networks to Model Commodity Market Price Fluctuations

While conventional neural network training algorithms produce static results given a constant input, we wish to use them to model a dynamic phenomena: commodity market price fluctuations. Recently, researchers in economic modelling and forcasting have been investigating the use of nonlinear dynamics to explain market forces. Through analysis of market data, they have been able to show evidence of chaotic behavior in these markets [Sav91][And88].

This section describes a technique to *model* the behavior of commodity market price fluctuations which is included to extend the modelling capabilities of the connectionist component of PSE. Minsky [Min86] defines modelling as follows: "Any structure that a person can use to simulate or anticipate the behavior of something else." The goal here is not to specifically predict market prices. Instead, we agree with Stern [Ste] that neural networks are not capable of accurately predicting the stochastic behavior of markets, but can be used to build effective models that aid in understanding the markets' behavior. Thus, models of the trends in pricing are chosen to improve understanding on how market prices fluctuate and how various commodities affect one another's prices.

An example of a commodity whose price fluctuates chaotically is the gold market. For example, if the jewelry market has increased demand, it will cause an increased demand for gold and silver. The increased demand for these metals will cause their price to rise in a chaotic manner as investors take profits. Eventually, the price will stabilize at a new level until new circumstances result in once again altering the price. Likewise, the rise in gold and silver will cause a relative rise in the price of platinum. Examples from actual market

data will be used to support this point.

Neural networks present a natural way of modelling the effects of price changes in such commodity markets, because if one commodity, say jewelry, has increased demand, so will other related commodities like gold. In the chaotic neural model presented in this chapter, neurons represent commodities and connections represent the effects they have on one another. A chaos-producing feedback equation is then introduced into the training algorithm to generate the effects of noise and dynamic behavior in the commodity prices to be modelled.

The utilities for the chaotic neural networks described in this section are contained in a PSE module, and the use of these utilities to model market prices is an important example.

7.4 Other Work on Chaotic Neural Networks

Chaotic neural networks have been designed based on backpropagation [Ai90] and have been applied to associative memory [Ik91]. However, the use of connectionism in this thesis is to represent higher level knowledge.

7.5 The Mechanism

Given a price associated with a commodity, the percentage of price change for a given day is estimated based on changes in other related commodities. The system models the perturbations in the price as time passes followed by a stabilization. Given the activation propagation algorithm as was presented in section 6.1.4:

$$a_i^{k+1} = a_i^k + \delta(a_i^k) sigm(\sum_j w_{ij} a_j^k) \tag{7.1}$$

The *sigm* function is a threshold function that uses truncation to keep the activity value boundaries between $[1, -1]$, and the δ function computes $1 - abs(a_i^k)$ to guarantee that there is no drastic change in the value when there is a value close to 1 or -1. Chaos is then introduced into the system by substituting X for $\sum_j w_{ij} a_j^k$ in the equation above and then feeding it through a chaos-producing function as follows:

$$\lambda X(1-X)\alpha + X(1-\alpha) \tag{7.2}$$

where $\lambda = 4$ and α has a value in $[0,1]$. α has a value that starts at 1 and gradually decreases to 0 over time to model the progression from unstable prices to stable ones. As each day goes by from a new instability in the market, the amount of chaos in price changes reduces. The gradual decrease in α models a system that starts off being entirely chaotic, then gradually moves towards one that is partially chaotic, and finally stabilizes.

Experiments have shown that values were not decreasing at a desirable rate, so the result was multiplied by $exp(-(k+1)/\theta)$ where θ is a constant value that can be modified to increase or decrease the leveling of prices. Thus, the final equation became:

$$a_i^{k+1} = a_i^k + exp(-(k+1)/\theta)(\lambda \sum_j w_{ij} a_j^k (1 - \sum_j w_{ij} a_j^k)\alpha + \sum_j w_{ij} a_j^k (1-\alpha)) \tag{7.3}$$

In the real data for precious metals, prices never fluctuated by more than seven percent in a single day. Thus, it was necessary to keep a range on the price fluctuations by taking the modulo β of the result, where β is the maximum percentage change for a single day's trading.

7.6 Example Network

The following code defines a chaotic neural network in PSE using non-persistent objects. The application describes the relationships between the precious metals: gold, silver, and platinum. *Defmodule* defines a module of code. *Defneuron* defines a neuron object which the system will translate into neurons in a network.

```
(defmodule metals
  (standard neu-cnn plists aux aux-macros)
  ()

(defneuron gold (newron)
  ((affects initform '((silver . 1.0) (platinum . 0.6)))))

(defneuron silver (newron)
```

```
  ((affects initform '((gold . .75) (platinum . 0.6)))))

(defneuron platinum (newron)
  ((affects initform '((gold . 0.45) (silver . 0.45)))))

(build-neural-network)

(set-same-ln '(affects . silver) '(is-a . silver))
(set-same-ln '(affects . gold) '(is-a . gold))
(set-same-ln '(affects . platinum) '(is-a . platinum))

(set-activity-level 'is-a 'gold -0.022)
(set-activity-level 'is-a 'silver -0.01)
(set-activity-level 'is-a 'platinum -0.015)

(compute-new-chaos-act 10)
)
```

Build-neural-network translates the objects defined by *defneuron* into a connectionist representation. The numeric values associated with the names indicate the values of the weights on the links between the neurons. Back links exist from the neuron class name (eg. *gold*) to the neurons created from the slot names (eg. *affects silver*). Figure 7-4 illustrates the links and neurons created from the above code. Solid lines represent links that have weights as defined by the user with *defneuron*, and the dashed lines indicate back links which represent upward inheritance with only fractional weights [Bur92]. Note that the function *set-same-ln* must be called to specify that links exist between neurons in a way that can not be adequately described in the *defneuron* primitive. Notice that the values for the weights are hardcoded, and were produced through observation and experimentation.

7.7 Real Data vs Generated Data

The graphs in Figure 7-5 are from four different time periods when there was a somewhat dramatic price fluctuation around the time of the gulf war (winter-spring 1991). Each interval indicates a day. Note that there is a dramatic rise or drop of prices. The graph is continued only to the point of stabilization. By observation of the graphs in figure 7-5, one can see that prices fluctuate chaotically. For example, in the second graph, there is a drop the first day of 7% followed by a one percent increase 2 days later which is then followed another drop. Also notice that the prices of the different precious metals rise and drop somewhat together. For example, in all the graphs of figure 7-5, the prices follow the

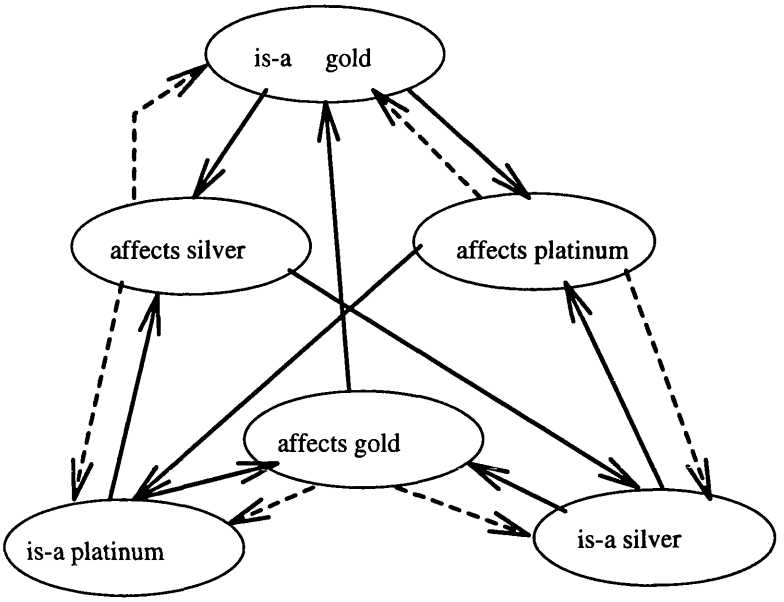


Figure 7-4: Internal Network Representation

same relative up and down movement.

The graphs in Figure 7-6 were produced from data generated from the chaotic neural networks described in section 7.6 where the change for the first day is the same in the corresponding graph of real data. The rest of the data, however, is generated by the chaotic neural network mechanism. Observation of the time history of the graphs indicates that the values generated are chaotic. Notice that while the match is clearly not exact, the chaotic fluctuations of the market are represented as is the effect of the different commodity prices rising and falling somewhat relative to one another.

While the numbers are not the same in the real data as compared to the generated data, it does aid as a first step in understanding the chaotic price fluctuations of related commodities. Further tuning of the chaotic neural network and analysis and understanding of chaotic market fluctuations should produce even better models.

7.8 Conclusions

This chapter contains extensions made to the POCONS connectionist simulator which was added to PSE to provide a simulation environment with support for knowlege-based consultation. A description of the SIMD and MIMD architectures and the implementation of POCONS networks on them were discussed. Results were presented for a 440 node connectionist network (an extension to the logic circuit one presented in the previous chapter)

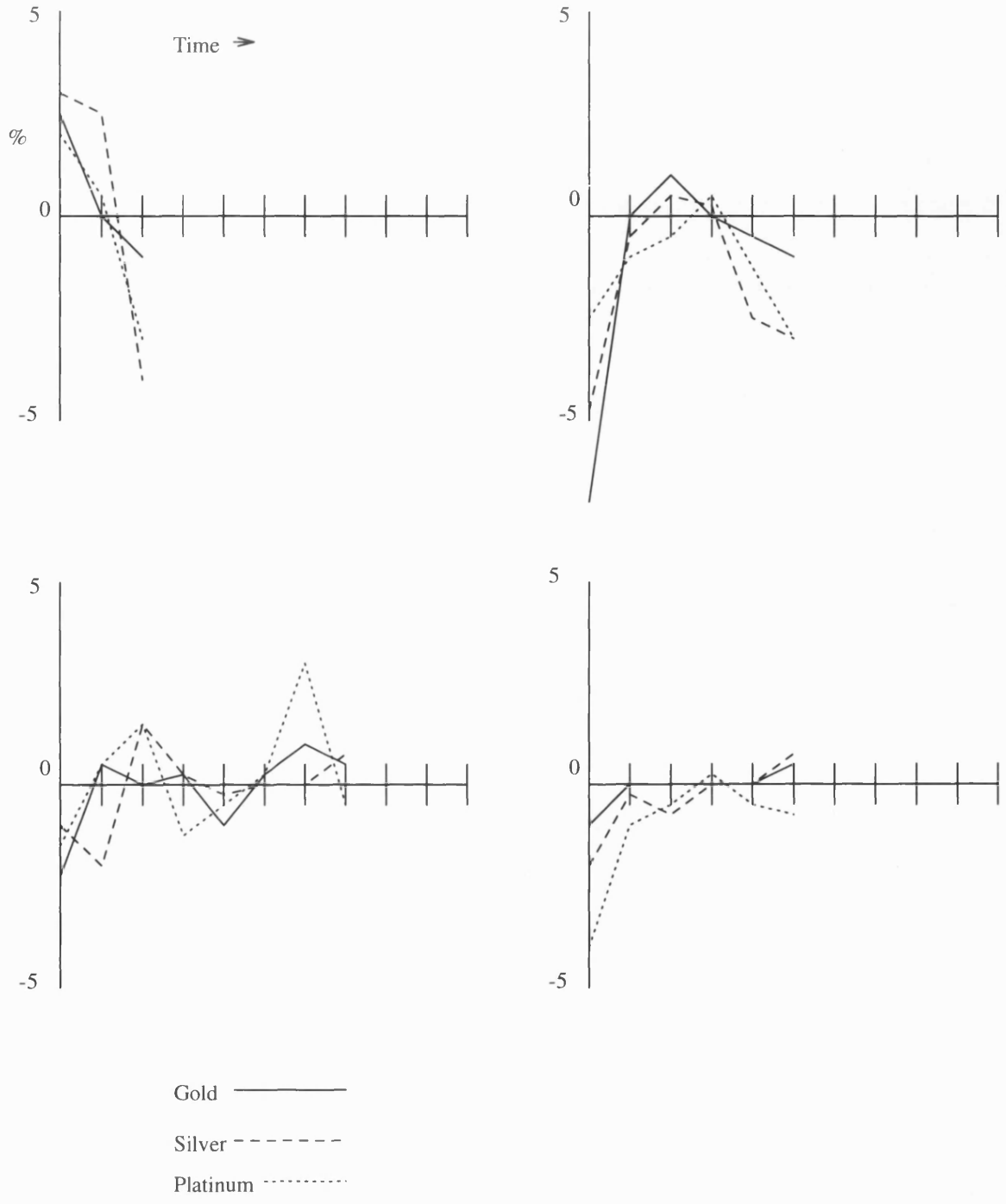


Figure 7-5: REAL DATA

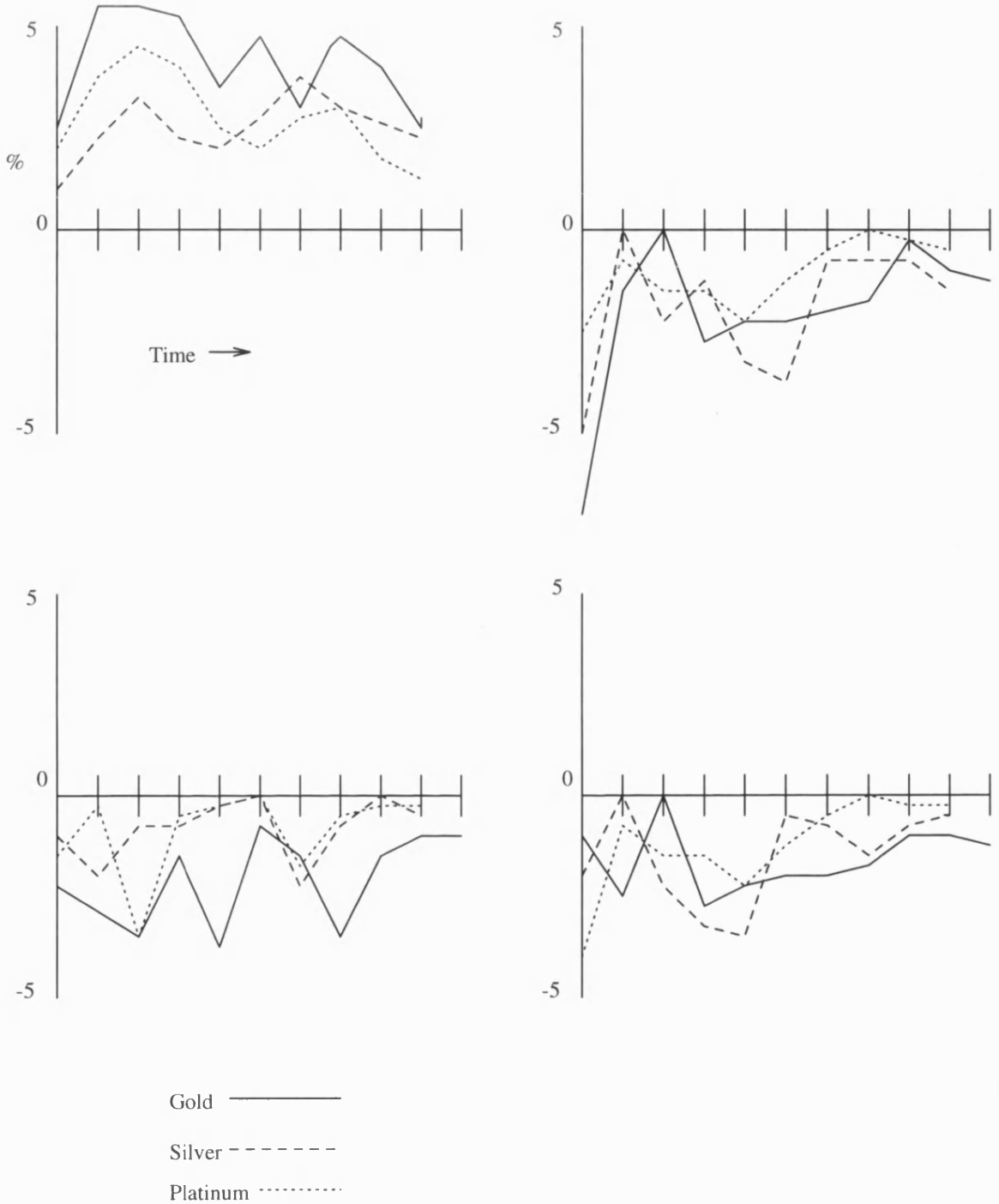


Figure 7-6: GENERATED DATA

executing on four processor Stardent achieving *more than double speedup.* This result shows that the explicitly parallel nature of connectionist networks can be practically exploited.

A built-in feature for storage and checkpointing using persistent objects has also been presented. Not only is it an elegant solution to the problem, but results in improved performance as well. The implementation of this feature has been described, and an example has been presented. Performance improvements resulting in double speedup have been reported using this facility for networks trained using activation propagation, and greater performance improvements will result from the storage and reuse of networks trained using more costly algorithms such as backpropagation.

Finally, there was a presentation on a mechanism, implemented in PSE, for modelling the behavior of commodity market prices using chaotic neural networks. First the author's own chaotic network mechanism based on activation propagation was described, and then an example was given of its usage for modelling the behavior of the precious metals market: first by showing and describing the code, and second by illustrating and describing the internal representation of the system. Finally graphs were presented which were generated by the system and were compared to graphs of real data. While the results are not the same, the model helps us further understand the chaotic price fluctuations of related commodities. It is convincing, however, that through tuning and modification of the parameters to the chaotic neural network, the model can be improved, because in theory, chaos can be used to model the behavior of commodity market prices. Neural networks are an effective tool for modelling in this case, because they are a natural way to describe the relationships between commodities and their prices.

While the model does not mirror the example data, it does show similar chaotic behavior. The model also shows inherited effects from one commodity upon another. Thus, the comparison of the model against the real world example suggests that by better understanding deterministic chaos and time series, one can better understand what to expect in the future of commodity market prices. This work is a first step in understanding this process, and while it does not tell us how to invest, it gives us an important experimental tool to help develop theories about the behavior of market prices.

In conclusion, parallel persistent object-based simulation has been applied to the connectionist paradigm. Parallelism has been shown to be advantageous for improving performance and persistence has been shown to have the advantage of providing an elegant means for reuse of neural network objects.

The next chapter will contain a discussion of Petri net simulation facilities added to PSE with applications that utilize parallel processing and persistence.

Chapter 8

Petri Net Modelling in PSE

The purpose of this chapter is to show that persistence and parallelism which were applied to process-based and connectionist simulation in previous chapters, can also be applied to Petri net simulation. To achieve this goal, this chapter covers the Per-Trans representation language that has been added to the EuLisp version of PSE to provide support for *both* higher level representations which can greatly simplify the modelling process, and hybrid models which are often required, as was shown in section 1.2, for the development of large-scale simulations. Per-trans is a representation language for the description of stochastic Petri nets. It makes use of persistent objects for the storage and reuse of the simulation objects and the simulation history. After describing Per-trans, an example will be presented. Then a technique will be described which vastly improves the performance (30-fold) of serial Petri net simulations using Per-trans. Results of the extension of a distributed virtual shared memory simulation will be presented, and it will be followed by a discussion of parallel techniques for Petri net simulation. Finally, results from actual parallel simulations will be presented as well.

8.1 Per-Trans: A Persistent Object-based Stochastic Petri Net Representation Language

Petri nets are widely used in the simulation of concurrent systems [Pet81]. As a result of the popularity of Petri nets, there have been a variety of tools developed [Fel89]. These tools allow graphical editing and creation of Petri net models. This chapter describes a tool for the development of Petri nets: a language called Per-trans. It features the fusion

of persistent object technology with Petri net development. Per-trans is a component of the EuLisp version of PSE which was added for this thesis. The motivation for PSE stems from the need to support large-scale persistent object-oriented simulations. The addition of Per-trans expands the application areas that PSE can be used to apply parallelism and persistence to.

Per-trans has features that simplify the task of developing stochastic Petri net models [Mar89]. It contains constructs that specify the places, transitions, and token locations in the Petri net. The underlying system handles all the procedural execution of the simulation. Per-trans allows Petri net components to be represented as persistent objects.

8.1.1 Per-Trans Components

Per-trans provides an application programmer with primitives to represent and execute simulations using the stochastic Petri net model. Since it is implemented within PSE, it contains all the advantages associated with persistence that exist for connectionist and process-based simulations as described in chapters 5, 6, and 7. Per-trans simulations contain objects for places and transitions. Theses objects are the two main components of a Petri net and the connections and delays associated with them define the Petri net model. The application programmer can decide whether he or she wants some or all of the net to be persistent.

Per-trans, like POCONS (described in chapters 6 and 7) is also declarative, which means that the programmer need not specify procedural information for the model or for the use of parallelism and persistence as supported by PSE. In Per-Trans the programmer need not specify any of the control information used to determine when a transition will fire and send tokens throughout the net. The Per-trans defining forms generate an event-based simulation that is executed by the underlying scheduler and simulator. The programmer need only specify the places and transitions, where they are connected, and any time delays that might exist on transitions (details of the language constructs will appear in section 8.1.4). The internal scheduler examines places and transitions to determine whether a transition is enabled and when it should fire. It also passes tokens to places that are enabled once a transition fires. The underlying scheduler sends messages to objects that contain a time stamp for when they should execute. It then executes those messages at the appropriate simulation time.

Also, Per-trans allows the application programmer to embed Lisp code in the definition of specific nodes (places and transitions). The embedded code will be executed when a token moves to the node's location in the network. Such embedded code can be used to process information or produce graphical output illustrating the net's behavior. Figure 8-3 is an example of graphical output produced in X-windows from a Per-Trans specification.

8.1.2 Stochastic Petri Nets

The basic concepts applicable to all Petri nets is that they are a graphical and mathematical modelling tool which can be used to simulate many different systems [Mur89]. They are a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic. As a graphical tool, basic Petri nets can be used as a visual communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems. Petri nets can be used by both practitioners and theoreticians.

A basic Petri net is a particular kind of directed graph, together with an initial state called an initial marking. The underlying graph of a Petri net is a directed, weighted, bipartite graph consisting of two kinds of nodes, called places and transitions, where arcs are either from a place to a transition or from a transition to a place. In graphical representation, places are drawn as circles, transitions as bars or boxes.

There are many varieties of Petri nets. One of the most useful varieties are called stochastic Petri nets (SPN)'s. SPN's are useful in performance evaluation of concurrent systems. The type of SPN that will be focused on at this point is called a deterministic stochastic Petri net (DSPN) [Mar87]. DSPNs contain transitions with deterministic (constant) delays, immediate delays, and exponential delays. The organization of a particular network represents the system being modeled. For example, figure 8-1 illustrates a multiprocessor system modeled using a DSPN. The processor active state (**p1**) indicates that the processor is ready to make a request of memory. Tokens residing at **p1** indicate free processors. At initialization time, the number of tokens residing on **p1** indicates the number of processors the system has in total. Since more than one processor can be free simultaneously, more

than one token can reside at **p1** at any given time. Transition **t1** represents a timed delay transition indicating that some processing time is required for the system to determine which memory to access. The place **p2** represents the state where the computer system being simulated must choose which central memory to access. The states represented by both **p3** and **p4** indicate that a memory request is being made to one of the central memories. Transitions **t4** and **t5** cause the tokens to leave **p5**, **p9**, and **p6** indicating that the bus and the central memory are busy. Transitions **t6** and **t7** are time delays modelling the time required to send information from memory to a processor. Finally, when the access is finished, the tokens are sent back to **p5**, **p9**, **p6**, and **p1** respectively indicating that the memory access has completed.

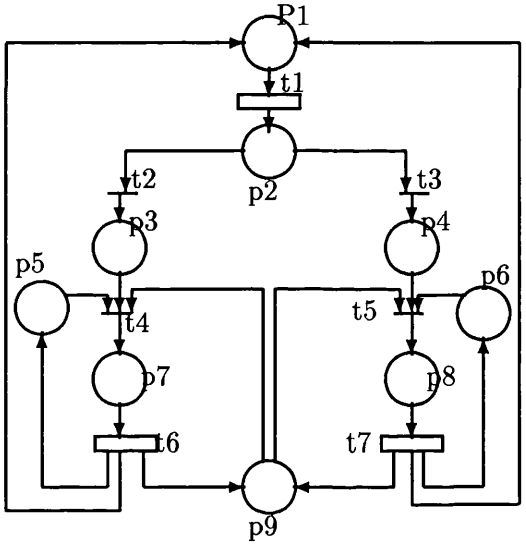


Figure 8-1: Petri net for Multiprocessor System

8.1.3 Petri Net Simulation

The algorithm for simulating a DSPN is [Mar86]:

```
loop for each  marking

0. Activity for each transition enabled is restarted for each marking.

1. If only one transition enabled
      then fire it
   else if several transitions are enabled in a marking and all transitions
```

```
        enabled are timed, then transition with shortest delay fires.
    else if (only one immediate transition is enabled)
            then fire the immediate transition
    else if (several immediate transitions are enabled)
            then choose the transition to fire based on priorities or probabilities.

end loop
```

This algorithm requires lock-step execution of transitions. Each time through the loop, the system computes the delay for each enabled transition. It then fires the enabled transition with the shortest delay and updates the places on the network accordingly. A *marking* is the current state of a network.

Note that in the case when several immediate transitions are enabled, the algorithm can resort to either using priorities or probabilities. When using priorities, it fires the enabled immediate transition with the highest priority. If using probabilities, it fires the enabled immediate transition with the highest calculated probability.

The requirement of this algorithm that only one transition be executed and then all enabled ones checked each time through the loop does not apply well to the requirement of having several processes that execute concurrently as is required in optimistic and conservative simulation techniques. Therefore, specialized techniques for exploiting parallelism in stochastic Petri nets were investigated which includes the parallel firing of transitions under certain constraints which maintain the consistency of the model. The description of these techniques and results obtained from their use will be described in section 8.3.

8.1.4 Defining Forms

The discussion will now move from the discussion of general Petri nets and SPN properties, to the basics of the Per-Trans language. Note the similarities in style between the defining forms in Per-Trans and those in POCONS. In both cases, the forms are EuLisp macros that expand into code that creates persistent objects to store modelling attributes and methods which perform modelling operations.

The three underlying classes of objects created by the Per-trans defining forms include: **petri-net**, **dbtransition**, **dbplace**, and **dbtoken** corresponding to the elements of a Petri net. It also contains the following defining forms: **defdb-petri-net**, **defdbtransition**, and

defdbplace.

defdb-petri-net requires a list of **places** and **transitions**. It allows a Petri net to be encapsulated within a single object.

The **dbtransition** class consists of the attributes **delay, places-in, places-out, tokens-entered, tokens-left,** and **token-delays**. The **delay** field stores values to be used as probabilistic or constant time delays. These time delays are located on transitions to delay their firing. The time delays are relative to simulation time–not real time. **Places-in** and **places-out** are slots which contain lists of places entering and leaving the transition. **Places-in** indicates to the Per-trans run time system which places have arcs entering the transition and can thus enable it for execution. Likewise, the Per-trans run-time system examines **places-out** to determine which place or places will receive tokens from the enabled transition. **Tokens-entered**, **tokens-left**, and **token-delays** are internal to Per-trans. They store audit information on the tokens entering and leaving a transition, the times they entered and left, and the delays associated with them. The user need only, however, use the **defdbtransition** primitive as follows (note that the syntax is Lisp-based):

```
(defdbtransition t1
   (delay poisson val) ;Either poisson, normal, or exponential.
   (places-in  list) ; A list of the input places.
   (places-out list)); A list of the output places.
```

The call to **defdbtransition** produces a generic function and a **dbtransition** object by automatically generating a call to make-instance as follows:

```
(setq transitions
    (cons (make-instance dbtransition
                         'name t1
                         'delay (poisson val)
                         'places-in list
                         'places-out list)
          transitions))
```

The above object is used by the system to store information about the tokens traversing the net. It also contains information about which places the transition depends on for firing and which places it enables after it has fired. Along with the object creation shown above, the following method definition will be generated by the underlying system as a result of the above execution of **defdbtransition** called on **t1**.

```
(defmethod t1 ((tr transition) (old-place place) (tok token))
    SET VALUES
    (apply work (delay tr))
    SET MORE VALUES)
```

t1 will be executed by the system when all of the input places to transition **t1** are enabled. The values that get set inside the method store data concerning the number and frequency of transition firings.

The construct for defining immediate transitions is called *defdb-imed-transition.* It is identical to *defdbtransition* execept that it has a *priority* and a *probability* field. During execution, when more than one immediate transition is enabled, Per-Trans checks whether the *probability* field is set to a non nil value on the enabled transitions. It then evaluates the contents of the slot and uses it to compare against the probablility it gets from the other slots. The application programmer must make certain that if one immediate transition has a probability set, that the others do as well.

Priorities work in a similiar manner. If the probability field is null, then the Per-Trans simulator looks at the priority field which has a default of one. It then compares the priority values for all the enabled immediate values and fires the one with the highest value.

Defdbplace works similarly except that there are no delays. Each place is represented by a generic function and an instance of the class **dbplace.** The attributes of the **place** class are as follows: **num-tokens, tokens-entered, tokens-left, tokens-allowed, transitions-entering,** and **transitions-leaving. Defdbplace** is used as follows:

```
(defdbplace p1
  (num-tokens num)) ;The number of tokens allowed to reside at a place.
```

As a result of the above code segment, **Defdbplace** will create a method of **p1** (from figure 8-1) which is similar in form to the one for **t1.** As in **defdbtransition, defdbplace** stores information on the tokens traversing the net. The instance contains information used by the system. Per-trans will execute the method attached to a specific place when it receives a new token. The slots for **trans-entered** and **trans-leaving** are initialised by the system based on the definitions in **defdbtransition** which indicate the **places-in** and **places-out** field. This feature keeps the programmer from coding redundant information.

When the Per-trans simulation executes, tokens get created when they move from firing transitions to newly enabled places. The underlying system will handle queues on transitions

which require more than one token to fire. It will also generate new tokens where there is more than one leaving a **place**. Per-trans allows for stochastic delays by accepting a time parameter in the **delay** field of a transition generated by any of the following probablistic distribution functions: **Poisson**, **normal**, and **exponential**. The programmer can also use constant delays (including zero) by putting a constant value in the **delay** attribute field.

To execute the Petri net, the programmer need only make a call to **(run-petri-net petri-net duration)**. The Petri net instance is returned by the call to **defdb-petri-net** and the duration is specified by the user to determine the length of simulation time that the Petri net should execute.

A portion of the Per-trans code for the Petri net of figure 8-1 appears below.

```
(defdb-petri-net mult-proc
  ((places p1 p2 p3 p4 p5 p6 p7 p8)
   (transitions t1 t2 t3 t4 t6 t7)))

(defdbplace p1
  ((tokens-residing 2)
   (code (lambda () (format t  "processor is active at time ~a~%"
                  (current-time))))))

(defdbplace p2
  ((code (lambda ()
         (format t "choosing one of the common memories at time ~a~%"
                   (current-time))))))

(defdbtransition t1
  ((places-in p1)
   (places-out p2)
   (delay normal 4 2)))

(defdbtransition t2
  ((places-in p2)
   (places-out p3)
   (delay normal 0 0)))


(defdbtransition t3
  ((places-in p2)
   (places-out p4)
   (delay normal 0 0)))
```

The call to **defdb-petri-net** indicates the different places and transitions in the system. It encapsulates the net in that several Petri nets can exist in primary memory at the

same time. By executing **run-petri-net** on the specified net, the system will execute a simulation of it. The calls to **defdbplace** indicate the tokens residing on the various places at initialization time. They also indicate the transitions connected to the place. **Defdbtransition** determines which places the transition is connected to. The **delay** field is required for the transition, because it determines the amount of simulation time that a passing token will be delayed before it can move on. In the case where the delay is **(delay normal 0 0)**, there is no time delay.

8.2 Techniques for Improving the Performance of an Object-Oriented Stochastic Petri-net Simulator

Numerical methods have been developed to evaluate the performance of SPNs, but require too much CPU time to evaluate large nets as are needed in industrial applications. Discrete event simulation provides the only alternative for solving these larger nets, because as the nets get larger, the complexity of solving them using discrete event simulation is significantly less than the cost of using numerical techniques.

The work described in this section has been concerned with making OOP technology efficient enough for use in simulating large Petri nets. In the initial implementation of PerTrans, the cost of object-oriented discrete event simulation was found to be significantly high due to method dispatch and slot-accesses. Also, list processing and garbage collection in EuLisp were found to be extremely expensive.

Thus, the system has been reimplemented with two levels. The top or interface level remains in EuLisp to allow users to define sets of Petri net objects and manipulate them in the EuLisp object system (called Telos). It also continues to allow Petri net simulation objects to be persistent. Thus, to the simulation programmer nothing has changed. Functionally, the system is the same; it has just been made more efficient, because the underlying simulator has been reworked.

The underlying or simulator level has been rewritten mostly in C and interfaced with Lisp. The system is composed of two layers as is shown Figure 8-2. The simulator level extracts the Petri net information from the Lisp objects and stores it in C data structures. It then executes the simulation in C using the techniques to be described which have reduced the execution time by 30 fold: the pure EuLisp version required 20 CPU hours to solve the ten page network, whereas after moving the simulator into C, it took only 35 minutes of CPU

time. These two layers give users the expressive power of a Lisp object system at the top level combined with an efficient low level implementation that has dramatically improved run-time performance. This resulting system produces the same analysis as numerical simulators, but in a fraction of the CPU time, which makes it efficient enough to simulate Petri nets which are too large for numerical simulators to handle [Lina].

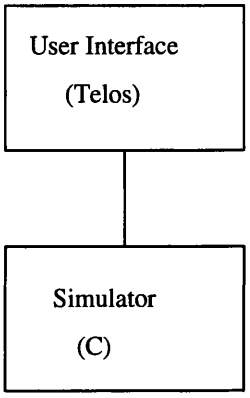


Figure 8-2: The User Interface and Simulator Layers

The four components which have been eliminated by the new implementation are as follows:

1. method dispatch
2. slot accesses
3. garbage collection
4. Lisp floating point arithmetic

The overhead of method dispatch was eliminated by rewriting all methods as C functions that expected specific arguments. The generic function lookups were replaced by direct function calls. Likewise slot accesses have been made more efficient by storing objects as C structures, and there is a separate array for storing all the objects in each class. Objects are then given an id number when initialized, and they are then referred to by their object identifier. When a slot access is made, it consists of a table lookup and a structure reference which is far more efficient than the dynamic type checking and slot descriptor lookup involved in Telos slot accesses.

The component that brought about the greatest improvement in performance was the movement of all list processing from EuLisp into C. The lists are now built dynamically

using *malloc* and *free*. As a result, the EuLisp garbage collector does not need to be used during a simulation. This modification alone provided an order of magnitude in speedup.

Finally, another factor which greatly improved the performance of the simulator was the movement of all floating point operations from EuLisp into C. The C functions do no type checking as Lisp would, and therefore the calculations are executed immediately. Also, the C compiler does good optimizations on floating point that EuLisp cannot forsee.

Therefore, by moving the simulation engine of the Petri net simulator into C, the performance of the system was improved 30 fold (20 hours before compared to 35 minutes after). It should be stated that EuLisp is still in the prototyping stages and should be much faster in later more stable implementations. Thus, the same comparison was done using CMU Common LISP to see what the performance improvements would be. The C version was still 10 times faster than the CMU Common Lisp version which is still a substantial amount.

It should be mentioned that these comparisons were used for non-persistent objects for both the EuLisp and EuLisp/C versions of Per-Trans. One could make the argument that the pure EuLisp version still has the advantage that objects can be made persistent by simply using **defdbplace** and **defdbtransition**. However, in the light of the information presented in the storage and reuse section of chapter 7, it is more sensible to store the representation information in persistent objects, while converting the network into an internal representation which will execute efficiently. Then when the simulation has ceased execution, the data in the internal representation can be checkpointed to the persistent objects by the system. Thus, as a result of the knowledge gained from the storage and resuse section, the EuLisp/C version is consistent with the idea of creating an efficient internal representation of the model. The information stored in the C data structures is accessable from EuLisp, so that information could be easily checkpointed and stored in the persistent Petri net objects.

8.3 Parallel Simulation of Stochastic and Colored Petri Nets

Now that an efficient means of applying persistent object-oriented simulation of Petri nets has been applied, it is now time to move onto the investigation of parallelism as applied to Petri nets for improving performance. This section describes several techniques that have been implemented for parallel simulation of stochastic Petri Nets [Mar89], and a parallel implementation of a Colored Petri net [Jen90]. The simulations were written in EuLisp [Pad91]

and executed on a three-processor Stardent Titan. Coarse-grain parallelism was utilized, because in preliminary experiments, fine-grained techniques actually reduced performance, and coarse-grain improved it.

One problem with the parallel execution of stochastic Petri nets is determining what to parallelize. Parallelism-related overhead must be small to achieve performance improvements. Thus, the standard parallel simulation techniques (optimistic [Jef85b] and conservative [Mis86]) have been considered and rejected by the author [Bur93b] [Bur93a], because while they provide elegant and general mechanisms, they tend to produce the best results on systems with coarse-grain events. At an invited talk, Brian Unger [Ung93] (who has a lot of experience in developing parallel simulations at Jade in Calgary, Canada) said that in Time Warp events need to be at least 200 milliseconds of computation to break even with overhead costs. 200 milliseconds is a vast amount of computation on todays super microprocessors. Unfortunately, the problem in the simulation of Colored and stochastic Petri nets is that events are very fine grain. Therefore, an application-specific approach similar to that used for the simulation of VLSI [Kar93] is employed which focuses on parallel aspects of the model rather than in parallelizing events.

VLSI circuit simulation exploits structural and temporal parallelism. Structural parallelism occurs through concurrent evaluation of two subcircuits which have no dependency relationship. Temporal parallelism occurs in subcircuits which have a dependency relationship, but the two subcircuits can be evaluated concurrently for different simulation instants. A high level pipeline is used so that subcircuit A can execute at time t_n while subcircuit B which feeds into A can execute at time t_{n+1}. This kind of model-based parallelism approach is taken in this chapter, because Petri nets contain independent subnets that can be executed in parallel in the same manner as VLSI subcircuits.

Stochastic Petri nets by definition can only execute a single transition per step. Thus, there is no inherent parallelism in the model. However, the techniques described in this chapter offer some ways around this problem. On the other hand, Colored Petri nets have been investigated, because their definition *does* allow them to execute concurrently enabled transitions in a single step. Thus, the Colored Petri net model has the advantage of at least some inherint parallelism. In addition to applying various approaches to parallel simulation of stochastic Petri nets, this section will discuss a parallel implementation which shows a reduction in the complexity in the simulation of Colored over stochastic Petri nets.

8.3.1 Stochastic Petri Net Application

The Petri net representation for a 10-page virtual shared memory (VSM) system is the application *testbed* for the parallel simulation algorithms. Figure 8-6 contains an X window screen dump of a graphical illustration of the one page version of the simulation. The X window graph animates the Petri net's execution by specifying the number of tokens on each place and updating them after a transition fires. When transitions fire, the graphical interface causes the boxes marking transitions and the lines pointing to the new place where the token will reside to flicker to indicate the movement of tokens.

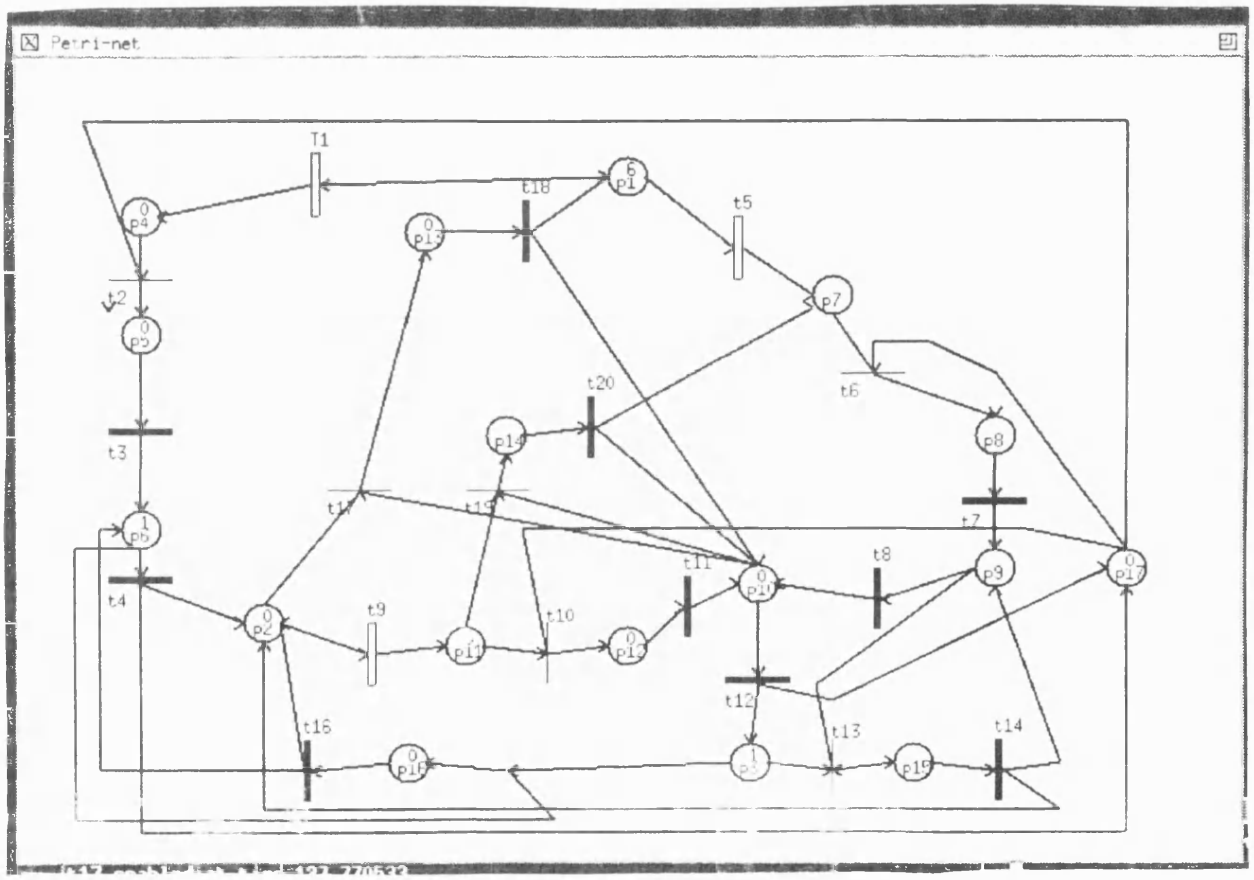


Figure 8-3: Graphic interface for one page VSM simulation

The Petri net model shown is based on a one page virtual shared memory system developed at GMD [Linb]. There are two kinds of parallelism being discussed here. The first is the simulation of a Petri net *model* which contains parallelism: a virtual shared memory system. The second is the parallel *execution* of that model to improve the performance of calculating its solution. The Petri net *models* a parallel system which is not to be confused with the parallel *simulation* of the Petri net itself.

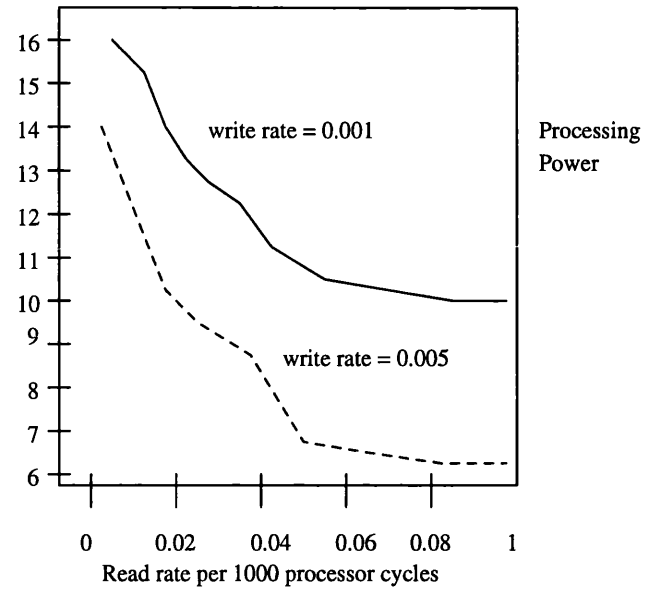
The Petri net model represents a protocol for guaranteeing sequential consistency in a VSM system. The strong consistency scheme allows a single-writer or a multiple-reader status for each shared page with a write-invalidate protocol. Comparable results for the one-page model were exhibited by *both* Per-Trans and the GMD numerical simulations[Linb]. The model was extended to ten pages by the author through the copying of the internal read-write access protocol portion of the net ten times and linking the copies with the single server and request nodes. As a result, a page access has ten pages to choose from instead of just one. The results for the ten-page system using Per-trans appear in Figure 8-4. The top graph in Figure 8-4 shows that using the model for a 16 processor machine (done by having 16 reside on P1 initially), that the read rate and write rate have a dramatic impact on the sytem's processing power. Processing power is the measure of the system's performance minus the system overhead. The simulation results show that lower read and write rates result in greater processing power – as would be expected. The *simulation* results for the ten-page model are similar to that from the one-page model. The only real difference is that the server utilization is slightly higher in the ten-page model at low read rates.

The lower graph in Figure 8-4 illustrates the utilization of the single file server for virtual memory paging, and the effect that different read and write rates have on it. The results show that lower read and write rates result in less utilization of the server – as would also be expected.

This ten-page model is too large to be executed on a numerical simulator [Lina], but can be simulated using the Per-Trans discrete-event *simulator*. The ten-page model was used to test the execution time in Per-Trans as will be shown.

8.3.2 Parallel Programming Construct

Futures, as was explained in section 7.2.2, were used on MIMD machines for parallel connectionist simulation. For the purposes of parallel Petri net simulation, a Futures package has been especially implemented in EuLisp by the author. Since the creation of separate threads is quite expensive, it creates several threads at start up time and recycles these threads, so that no new ones need to be created. If all of the Futures are in use and a request is made by the application program for another one, the request will block until a currently used Future is freed and can be made available for the new request. Please note



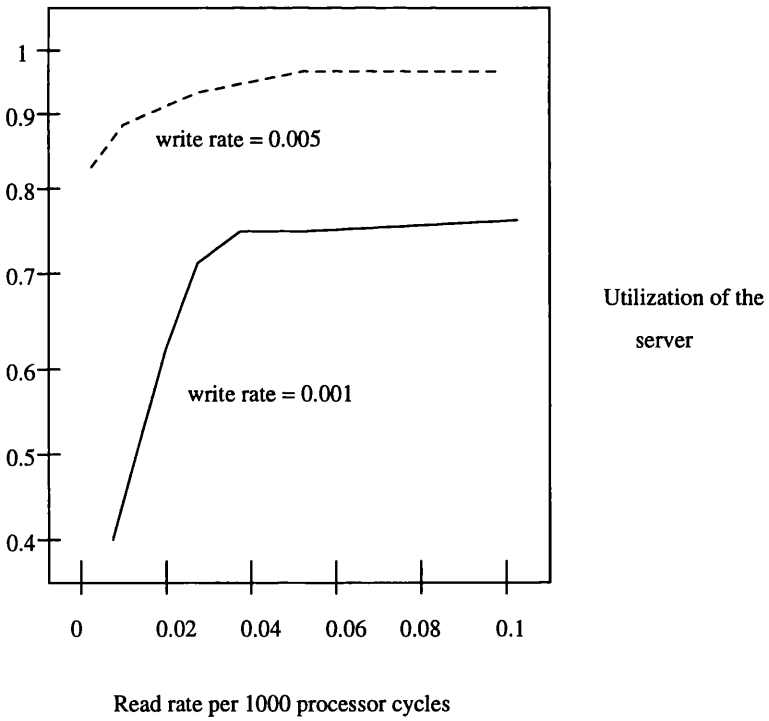


Figure 8-4: Simulation Results from 10-page DVSM Petri Net

that the reuse of threads wasn't necessary for MIMD connectionist simulations, because only three and four Futures were utilized and none were reused. However, the techniques used for parallelizing Petri nets in this thesis can require reuse of threads, because it is possible that a Future may be used and freed up many times.

8.3.3 Parallel Replication

Parallel replication was straightforward to implement and provided impressive results. Replication is based on the requirement of the simulation algorithm that three generations of X number of transition firings occur where X transition firings cause the entire net to be traversed by tokens a multitude of times. Each generation has a different seed to the random number generator. The parallelization has three generations of the Petri net simulation executing in Futures in parallel. Data is collected separately for each generation. Once each generation has finished execution, the system collects data from the different generations and merges it. The merging process is serial, but consists of a minor amount of the total computation of the simulation, so it does not have much of an impact on performance. As is shown in figure 8-5, experiments using this technique on three processors have shown a substantial improvement of performance by a factor of more than double. Figure 8-5 contains a graph illustrating the results from executions using Futures on one, two, and three processors compared to the serial non-Futures version executed on a single processor. Notice that the single processor Futures version takes only a small percentage more time to execute than the serial version. This result can be accounted for the limited number of Futures (3) and the efficiency of the implementation of the author's Futures package. While replication may not be the most intellectually exciting method for parallel simulation, it is straightforward to implement and produces good results.

8.3.4 Dependency-based Parallelism

Another technique has been implemented and tested for parallel execution of stochastic Petri nets which is based on dependencies. Dependency-based parallelism is similar to structural parallelism for VLSI circuit simulation mentioned in section 8.3. Under the dependency scheme, transitions belong to the same set if they share any common incoming places – transistions linked to the same incoming arc belong to the same set even if they are linked to other places as well. The reasoning behind this method is that when a place

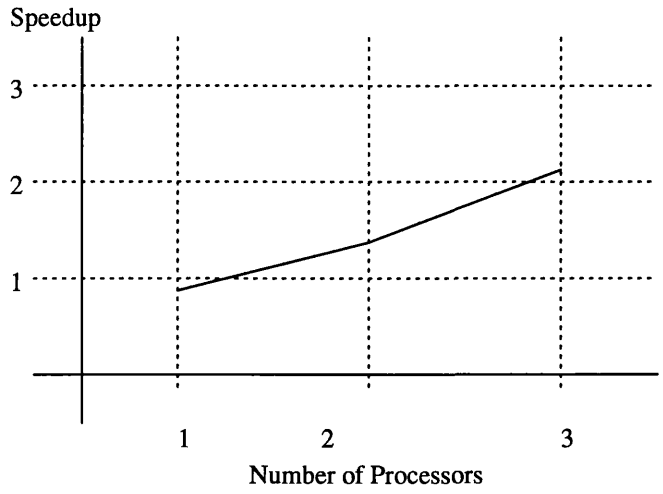


Figure 8-5: Time taken for the Per-Trans *simulator* to execute the 10-page DVSM model on multiple processors (using parallel replication) vs. a single processor (using the sequential algorithm of section 8.1.3

is enabled, it will enable (on its own or in conjunction with other places) all the transitions it has outgoing links to. Transitions that are concurrently enabled and are not members of intersecting sets can be fired concurrently, because they are enabled by different places. Figure 8-6 illustrates this technique.

In Figure 8-6, there are two dependency sets. The transitions in set A form a dependency, because *t1*, *t2*, and *t3* are dependent on *p1*; likewise *t3* and *t4* are dependent on *p3*. Thus, *t4* must be a part of the dependency set, because it cannot execute concurrently with *t3*, and *t3* cannot execute concurrently with *t4*. If they were executed concurrently, it would be a violation of the semantics for a Petri net which says that one token can cause only a single transition to fire – not many. Any one of the transitions in set B can fire concurrently with anyone of the transitions in set A, because there are no dependencies between the two sets.

The clock is then updated to the current time *plus* the maximum delay of all the transitions being fired to represent the behavior of the simulator – with several transitions firing concurrently the time required for their execution will be the maximum it takes a transition to fire. If several transitions that are members of intersecting sets are enabled, then the system executes the one with the shortest delay only: if A and B are enabled transitions where $A \in X$ and $B \in Y$ and $\phi \neq X \cap Y$ then A and B can both execute when both transitions are *not* members of both X and Y.

Note that this technique is only useful for transitions with delays > 0. If there are any transitions with immediate delays enabled, then only one of them will get fired across the

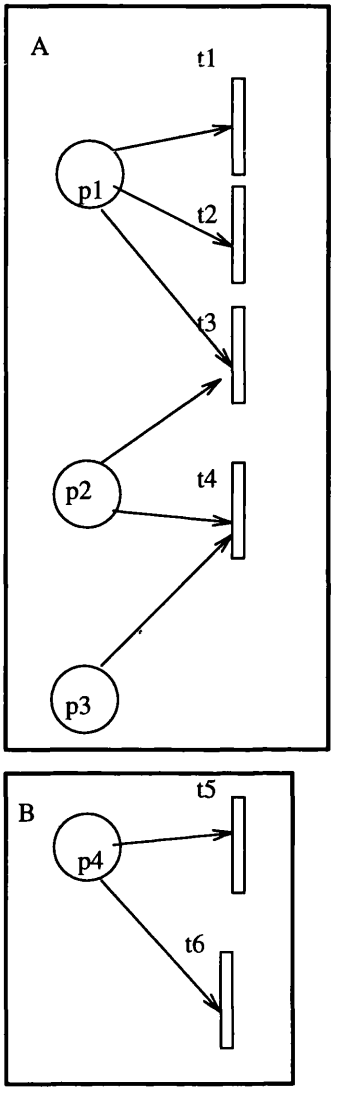


Figure 8-6: Sets of dependent transitions

entire net to maintain the semantics of the network.

A mechanism for dependency-based parallelism has been implemented and tested. On three and four processors, it achieved approximately 22 % speedup over the sequential simulator described earlier in this chapter. It should however perform better under a larger model containing large independent subnets (eg. 30-50 transitions).

8.3.5 Selection-based Parallelism

The dependency-based technique produces parallelism at the transition firing level. However a more significant portion of computation is required to select the next transition to fire than is required by the event which actually fires the transition. The stochastic Petri net simulation algorithm (listed previously) requires that the delay time for every enabled transition be computed for all enabled transitions to determine which transition to fire (see section 8.1.3). In some cases the calculation of delay can be a computation of *exponential distribution* which requires several floating point operations and a significant amount of CPU time. The algorithm must then search through the enabled transitions to find the one with the lowest delay-time. Thus, a selection-based technique is employed which partitions the enabled transitions into three threads so that their delay times can be computed in parallel.

The technique was straightforward to implement and when tested with the ten-page model produced about a 30% performance improvement over the serial version. However, as in the case of the dependency-based method, through simple analysis one can see that it is also well-suited for big (1000 node) Petri nets where large numbers (> 25) of transitions are concurrently enabled. The computation time to compute the delays for a large number of transitions then becomes a much more significant percentage of overall computation time.

8.3.6 Colored Petri-nets

Due to the difficulties in applying parallelism to SPNs, the author decided to investigate Colored Petri nets (CP-nets) [Jen90] to determine if they could be exploited for parallelism. CP-nets allow the modeller to make much more succint and manageable decisions than stochastic Petri nets allow without losing the possibility of formal analysis. The *color* in CP-nets refer to the model's support for different token types. Transitions can then have guard functions which require tokens of certain types to become enabled. The complexity of

a model can be divided between the net structure, the net inscriptions, and the declarations meaning that it is able to handle the description of much larger and more complex system. Using CP-nets, one can describe simple data manipulation (like the addition of two integers) by means of arc expressions (such as $x+y$) – instead of having to describe this by a complex set of places, transitions, and arcs. Figure 8-7 contains a non-hierarchical CP-net which describes a system where a number of processes compete for some shared resources. In the CP-net, places and their tokens represent states, while the transitions represent state changes.

In Figure 8-7, the constraints on each line determine what tokens, if any, pass along an arc. Likewise, guard functions (eg. (x,i) where x must be bound to a p or q resource and i indicates the number of resources) must be satisfied for a transition to be enabled. As a token or set of tokens traverse a CP-net, they carry contextual information collected on their journey indicating where it has been and the associated semantics. For example, in Figure 8-7 i gets incremented everytime a token moves from *T5* to either B or A. Note that according to the constraints in the figure, if the token is of type q, it moves to A. Otherwise, if the token is of type p it moves to B. The other constraints work similarly. For example on the arc from S to *T2*, the constraint states that if the token on B is of type p, then two e tokens should be removed from S when *T2* fires. Otherwise, if the token is of type q then only one e token should be removed from S to *T2*. The *init* values for A, R, S, and T represent the tokens residing on those places at startup time for the simulation.

The main reason for discussing CP-nets in this chapter is that they have the ability to execute *concurrently enabled* transitions in the same *step* without altering their semantics. Stochastic Petri nets are unable by definition to execute concurrently enabled transitions in the same step and thus modifications to the simulation algorithm must be used to support parallelism.

However, CP-nets explicitly allow concurrent execution of transitions in the same step and therefore provide a much more elegant model for implementing parallelism. *The CP-net model in Figure 8-7 has been implemented and execution of concurrently enabled transitions in the same step has been exploited in that implementation.* The algorithm is as follows:

```
loop
  determine-enabled-transitions
  fire-all-enabled-transitions-in-parallel
```

```
end-loop
```

The implementation was much simpler to develop than the dependency-based SPN model, and it shows a great amount of promise for significant speedup in larger models. The experience gained from experimentation in this domain suggests that when given a large model, examine it for independent subnets. If there is a significant number of independent subnets, it should be worthwhile to use the dependency-based technique. A large-scale colored Petri net model was not available, and the development and simulation of such a model is a major project and beyond the scope of this thesis.

8.4 Conclusions

Per-trans is a Lisp-based representation language which utilizes persistent objects and parallelism in simulations of stochastic Petri nets. It was added to PSE to provide a higher-level representation language which, as was shown in section 1.2, can simplify the modelling task. Per-Trans was also added to PSE to provide support for hybrid models which, as was shown in section 1.2, are often required for large-scale simulations. Probablistic time delays may or may not be embedded into transitions by the user. Per-trans is declarative in that the control of the firing of transitions and enabling of places is handled by the underlying system. The programmer can, however, embed Lisp code into the places and transitions to process information or produce graphical output to illustrate the behavior of the Petri net simulation. Per-trans is implemented in Eulisp under FEEL [Con] which has an X windows interface that supports graphical output and can be used in Per-trans applications.

To simulate large Petri nets, the system had to be made more efficient. Thus, the majority of the Petri net simulation engine was coded in C. The EuLisp interface remained the same, so that objects could be stored and manipulated as before. The only difference being that the Petri net was represented internally in C data structures. Thus, there were no more generic function calls, and slot accesses where reduced to table lookups. Also, dynamically created lists were *malloc*'d and *free*'d which eliminated the need for garbage collection. This technique improved performance some 30 fold. It should be noted though that EuLisp is still in the prototyping stage, and that once the design is settled, more efficient implementations can be expected.

This chapter has also covered techniques for the parallel simulation of Petri nets. The motivation for this work was to decrease the cost of solving Petri net models. The first paradigm

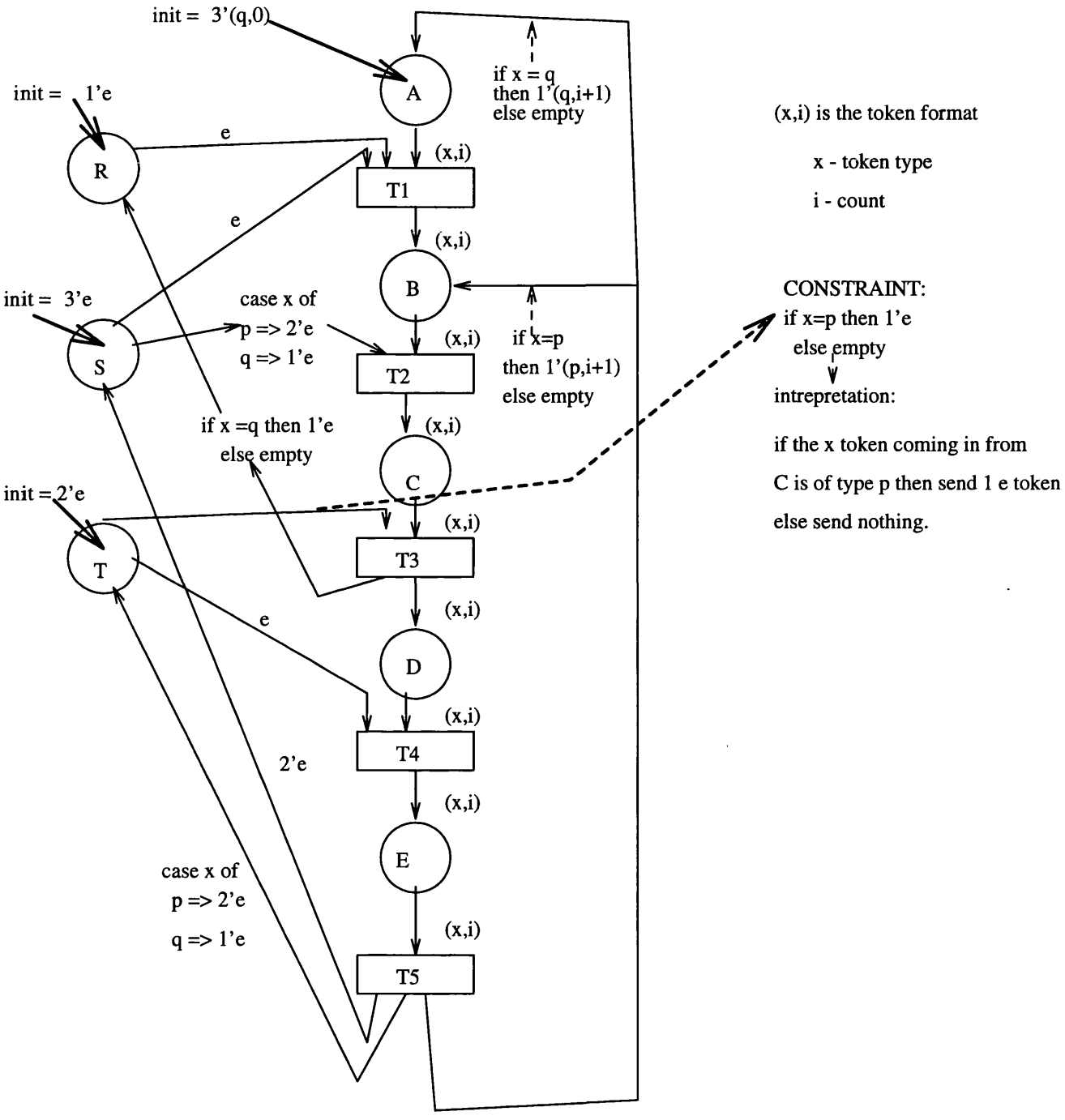


Figure 8-7: Colored Petri Net

inspected was stochastic Petri nets. Among the techniques tested in this paradigm, parallel replication has been shown to give the most impressive results by producing more than double speed up on three processors. Parallel replication was straightforward to implement, and even though it is not the most intellectually interesting technique, it does produce good results. Dependency-based and selection-based schemes have been implemented and tested and appear to be well suited for large applications. It would be worthwhile to investigate the advantages of these techniques on larger models in the future.

Colored Petri nets have also been investigated and they have the advantage of being explicitly parallel through allowing concurrent execution of transitions. The evidence shows that further investigation of parallel simulation of large Colored Petri nets holds great promise.

The knowledge gained from this work leads to the following design criteria when simulating Petri nets. First, use parallel replication. It is simple to implement and gives good results. If the net is large, inspect the independent components of it. If there are large components that are independent of one another use a dependency or selection-based scheme for a stochastic Petri net and concurrent execution of transitions for a Colored Petri net.

The underlying message expressed by this work is that Petri net simulation can be supported to utilize persistent objects and parallelism, much in the same way it was done for connectionist and processed-based simulation in chapters 5, 6 and 7. All represent the main simulation components (processes, neurons, places, and transitions) as objects. The Per-Trans language provides declarative constructs for the development of stochastic Petri nets (as does POCONS for connectionism – see section 6.1.3).

Chapter 9

Conclusion

This thesis has been concerned with the merging of parallelism and persistent objects in the development of a simulation environment. Parallel persistent object-oriented simulation has been applied in this thesis to several models to demonstrate its usefulness and to provide support for large-scale simulations which, as was shown in section 1.2, can require knowledge bases for consultation and may utilize higher-level representations like Petri nets, because they simplify the modelling process. Built-in defining forms and utilities that have been implemented for this thesis, which support these models using persistence and parallelism, have been described. Also, large-scale applications were implemented using these forms and utilities, *and* performance evaluations were presented which illustrate impressive results from both improvement of sequential codes and parallelization. Persistence has shown itself to be useful not just for perusal of objects after simulation completion, but also for storing simulation data that can be reused at a later date as described for connectionism in section 7.1.

The name of this simulation package that these forms and utilities were integrated into is the Persistent Simulation Environment (PSE) which was originally developed in CLOS, but was ported to EuLisp as a part of the work on this thesis. The port to EuLisp was made so that its support for parallel programming could be exploited. The EuLisp port provided several problems due to its lack of dynamic binding and module restrictions. While these problems were overcome, it was not possible to do so in an entirely transparent manner. The result was to make PSE a less than seamless interface between the application program and the database. To overcome this problem, one would have to implement persistence at the lowest level of EuLisp, so that persistent classes get instantiated across modules and

bindings of slot accessors to names could be done at run time. However, there are some serious ramifications resulting from such modifications. The compiler would have to be modified to handle the compilation of methods where the class is not present. The result would be a less-restrictive EuLisp. It would be, however, a EuLisp that does not meet the design goals of the EuLisp committee which is geared towards lexical instead of dynamic binding and non-mutually referential modules.

As a means to improve the performance of PSE's persistent object management system, a performance evaluation was done in chapter 3 comparing various object replacement algorithms, because for maximal improvement of performance, it makes sense to have the serial code be efficient before making it execute in parallel. An algorithm developed by the author (Faults Out) was found to perform best and was therefore the one added to PSE.

Event and process-based simulation utilities are necessary for simulation programming. Thus, as a basis for simulation and modelling, PSE was designed to augment a contemporary object-oriented language with discrete-event and process-based simulation facilities equaling those found in Simscript and Simula, and to tightly couple an object-oriented simulation language with a secondary storage facility to achieve the persistence of simulation objects. Chapter 4 described the simulation utilities for event and process-based simulation available in PSE. Examples of their use were also presented.

To provide the benefits of performance, object reuse, and perusal, parallelism and persistence were merged to support the process-based simulation for this thesis described in chapter 5. Optimistic and conservative models were compared, and the conservative technique was chosen, because since it ensures that no rollbacks will occur, it reduces the amount of memory usage and/or secondary memory modifications that would be required under an optimistic mechanism. Persistent objects were used to represent executing processes. This use of persistent objects simplifies the storage of the simulations' history. An example simulation was implemented to test the mechanism. It was an assembly line model, which was later extended to allow objects to be cloned which required modifications to the conservative concurrency protocol being used. The dynamic nature of Lisp greatly simplified the implementation of the cloning facility.

Chapters 6 and 7 presented a new component added to the EuLisp version of PSE, for this thesis, which supports connectionist simulation. The POCONS (Persistent Object-based CONnectionist Simulator) representation language was added to PSE to provide support for knowledge bases used in conjunction with simulations for planning and decision-making

as described in section 1.2. Persistence and parallelism were merged to support the model. The basic system allows the definition of neurons as objects which may or may not be *persistent.* The system was then extended to allow rules about the neuron elements to be specified. This rule-base facility is a basic building block for the development of knowledge bases. A large application was implemented which does a circuit analysis on the components of a flip-flop and a discussion was presented on how POCONS could be used for consultation of a knowledge base in a simulation. Then parallelism was added for these connectionist simulations on SIMD and MIMD machines to improve its performance. Improvements of more than double speedup were shown on four processors using the logic circuit application. Also a facility was added to POCONS for the storage and reuse of persistent neural network objects. The facility converts an internal neural network structure to persistent objects storing all information associated with the neural network, so that it can be recreated from storage. This facility was shown to have practical application as an elegant means to avoid retraining neural nets. Finally, the inferencing mechanism was also modified to promote chaotic neural networks. An application using the chaotic neural network facility was then implemented. The application modeled price fluctuations in the precious metals market. The neural network was not able to forecast the behavior of the markets, but it did show similar behavior. Further tuning of the model should produce better results, and while it would never be expected to model it exactly, it does aid in developing an understanding of the behavior of chaotic systems. The chaotic paradigm further expands the kinds of modelling that PSE supports which, as was shown in section 1.2, is required for the development of large-scale simulations.

Another module that was added to the EuLisp version of PSE as a part of this thesis was a Petri net representation language called Per-Trans. It was added to PSE, because as was shown in section 1.2, high-level representations like Petri nets can simplify the modelling process and large-scale simulations can require the combination of several models. Per-Trans allows the definition of Petri net components as objects which, like POCONS, may or may not be *persistent.* The base classes are *Place* and *Transition.* They serve as the basic building blocks for Petri net models. The focus of Per-Trans was on stochastic Petri nets, because they have been shown to be practical for use in performance evaluation of parallel and/or distributed computer systems. Per-trans was tested on several Petri net models including a 10 page distributed virtual shared memory model that was extended by the author from one that was designed at GMD in Berlin. Results from the simulation

were presented. Parallelism was utilized to support concurrent execution of stochastic Petri nets. Several techniques were tested and parallel replication produced the best results (more than double speedup on three processors). Though a case was presented that the others may give good results on larger nets. Finally, Colored Petri nets were investigated and an implementation showed that they were relatively straightforward to map onto a parallel machine. Transitions in Colored Petri nets can require complex decisions where it may be useful to consult a knowledge base, which could be built using POCONS, for advice.

The result of this work has been to show that various areas of advanced computing (persistent object systems, parallel computing, and high-level representations) can be integrated effectively and are useful in supporting various kinds of modelling. The advantage of having persistent objects in a simulation environment is that it gives the user the ability to do further analysis of the objects' stored results following program termination. Parallelism has been shown in this thesis to improve the performance of simulations.

The PSE system is not in a state to be used in a production environment, because it is essentially a prototype system written in a prototype system (EuLisp). Also EuLisp's module system and binding mechanism makes persistence awkward at best. Nevertheless, the applications that execute under PSE show that the ideas presented in this thesis have practical usage: it supports large-scale simulations by supporting the merger of several models, parallelism improves performance, and persistence provides an elegant means for storage and reuse of simulation objects.

The next stage of development would be to reimplement PSE for industrial purposes. Persistent object systems have progressed rapidly in the last four years, and are now ready for industrial use. I would pick one of the better ones (eg. Gemstone [But91]) and use its persistent capabilities. Then I would rewrite the utilities for discrete-event, process-based, connectionist, and Petri-net based simulation as are described in this thesis using C++ and Gemstone. The techniques described in chapters 6 and 7 for execution of connectionist models on a MIMD machine and parallel replication of Petri nets should also be utilized, because both techniques produced impressive results. The result would be a major advancement over any other simulation package currently available.

Appendix A

Glossary

- primary memory – The Lisp Image
- secondary memory – The database
- virtual image – The Lisp Image
- PSE – The Persistent Simulation Environment
- POCONS – Persistent Object-based Connectionist Simulator
- Per-Trans – Petri net simulator library for PSE
- Petri Net – A simulation model consisting of places, transitions, and arcs
- Connectionism – A knowledge-representation technique that relies on weighted links between facts
- Discrete-event simulation – a paradigm driven by a clock that advances at event-driven lump intervals as opposed to continuous time.
- Instantiation – the creation of an object or class.
- Persistent Objects – Objects that reside in the database as well as in the Lisp system and are managed by an underlying system instead of by an application programmer.
- Modsim – a simulation language that is sold commercially and is being extended in research laboratories.

Bibliography

[Ai90] Aihara, K., Takabe, T., and Toyoda, M. Chaotic Neural Networks. *Physics Letters*, 144(6-7):333–340, March 1990.

[Ik91] Ikeguchi, T., Adachi, M., and Toyoda, M. Chaotic Neural Networks and Associative Memory. In *Lecture Notes in Computer Science 540: Artificial Neural Networks, IWANN '91*, pages 17–24. Springer-Verlag, 1991.

[Abe93] Abellard, P., et. al. A Simulation of Artificial Neural Networks with Hypercard for the Scheduling of Data Flow Petri Nets. In *Proceedings of the 1993 European Simulation Multiconference*, pages 245–249. Society for Computer Simulation International, University of Ghent, Belgium, 1993.

[Ali93] Ali, L. A Connectionist Expert System With Cellular Atrophy For Process Planning. In *Proceedings of the 1993 European Simulation Multiconference*, pages 652–655. Society for Computer Simulation International, University of Ghent, Belgium, 1993.

[And76] Anderson, J. R. *Language, Memory, and Thought.* Lawrence Erlbaum Associates, Publishers, 1976.

[And88] Anderson, P., Arrow, K., and Pines, D. (eds.). *The Economy as an Evolving Complex System.* Addison-Wesley, 1988.

[Atk83] Atkinson, M, et. al. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, 1983.

[Atk89] Atkinson, M. and Morrison, R. Persistent System Architectures. In *Proceedings of the Third Annual Conference on Persistent Object Systems*, pages 9–28. Springer-Verlag, 1989.

[Bai89] Bailey, P. J. Performance Evaluation in a Persistent Object System. In *Proceedings of the Third Annual Conference on Persistent Object Systems*, pages 289–299. Springer-Verlag, 1989.

[Bal92] Balducelli, C. and Vicoli, G. Early Prediction of Electrolytic Cells Fault Conditions using Knowledge-based Simulation. In *Proceedings of the 1992 European Simulation Multiconference*, pages 641–645. Society for Computer Simulation International, University of Ghent, Belgium, 1992.

[Bla90] Blank, T. The MasPar MP-1 Architecture. In *IEEE CompCon*, 1990.

[Bob88] Bobrow, D., et al. Common Lisp Object System Specification. X3J13 Document 88-002R, 1988.

[Bou92] Boukachour, J. Job Shop Simulation Based on Object-Oriented Programming. In *Proceedings of the 1992 SCS Western Multiconference: Object-Oriented Simulation*, pages 8–12. Society for Computer Simulation, San Diego, CA, 1992.

[Bro85] Brownston, L., Farrell, R., Kant, E., and Martin, N. *Programming Expert Systems in OPS5: An Introduction to Rule-based programming.* Addison Wesley, 1985.

[Bur89] Burdorf, C., Gates, B., and Marti, J. Design, implementation, and performance of the RAND time warp system. Report in preparation, The RAND Corporation, 1989.

[Bur90] Burdorf, C. and Marti, J. Non-Preemptive Time Warp Scheduling Algorithms. *Operating Systems Review*, pages 7–18, April, 1990.

[Bur92] Burdorf, C. POCONS: A Persistent Object-based Connectionist Simulator. In *Proceedings of the 1992 SCS Western Multiconference: Object-Oriented Simulation*, pages 88–93. Society for Computer Simulation, San Diego, CA, 1992.

[Bur93a] Burdorf, C. and Fitch, J. Cloning Persistent Objects Under a Conservative Mechanism for Concurrency Control. In *Proceedings of the 1993 European Simulation Multiconference*, pages 598–602. Society for Computer Simulation International, University of Ghent, Belgium, 1993.

[Bur93b] Burdorf, C. and J. Marti. Load Balancing Strategies for Time Warp on Multi-User Workstations. *The Computer Journal*, 36(2):168–176, April 1993.

[But91] Butterworth, P., Otis, A., and Stein, J. The Gemstone Database Management System. *Communications of the ACM*, 34(12):64–77, October 1991.

[Cam91] Cammarata, S. and Burdorf, C. PSE: An Object-Oriented Simulation Environment Supporting Persistence. *The Journal of Object-Oriented Programming*, 4(6):30–40, October 1991.

[Cha81] Chandy, K. M. and Misra, J. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *ACM Communications*, pages 198–206, April 1981.

[Con] Concurrent Processing Research Group, University of Bath. FEEL: An Implementation of EuLisp – University of Bath, School of Mathematics, Unpublished.

[D'A88] D'Autrechy, C., Reggia, J. A., Sutton, G. G., and Goodall, S.M. A General-Purpose Simulation Environment for Developing Connectionist Models. *Simulation*, 51(1):5–19, July 1988.

[Dah66] Dahl, O. J. and Nygaard, K. SIMULA - an Algol-Based Simulation Language. *Communications of the ACM*, 9:671–678, 1966.

[Dah67] Dahl, J. and Nygaard, K. Simula: A language for programming and description of discrete event systems. User's manual, Norwegian Computing Center, 1967.

[Fel82] Feldman, J. A. and Ballard, D. H. Connectionist Models and Their Properties. *Cognitive Science*, 6, April 1982.

[Fel88] Feldman, J. A., Fanty, M. A., and Goddard, N. H. Computing with Structured Neural Networks. *IEEE Computer*, pages 91–103, March 1988.

[Fel89] Feldbrugge, F. Petri Net Tool Overview 1989. In *Lecture Notes in Computer Science: Advances in Petri Nets 1989*, number 424, pages 151–178. Springer-Verlag, 1989.

[Flo90] Floreen, P., Myllymaki, P, Orponen, P., and Tirri, H. Compiling Object Declarations into Connectionist Networks. *AICOM*, 3(4):172–183, December 1990.

[Fuj90] Fujimoto, R. Parallel Discrete-Event Simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[Geh86] Gehani, N. *Specifications: Formal and Informal–A Case Study: Software Specification Techniques.* Addison-Wesley, 1986.

[Gol83] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Gou88] Gould, R. *Graph Theory.* Benjamin/Cummings, 1988.

[Hal85] Halstead, R. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program Lang. Syst.*, 7(4):501–538, 1985.

[Hat91] Hatono, M., et. al. Modeling and On-line Scheduling of Flexible Manufacturing Systems Using Stochastic Petri Nets. *IEEE Transactions on Software Engineering*, 17(2):126–131, February 1991.

[Heb49] Hebb, D. O. *The Organization of Behavior.* Wiley and Sons, 1949.

[Her92a] Herring, C. ModSim A New Object-Oriented Simulation Language. In *SCS Western Multiconference: Object-Oriented Simulation.* Society for Computer Simulation, San Diego, CA, 1992.

[Her92b] Herring, C. and Whitehurst, R. A. Adding Persistence to an Object-Oriented Simulation Language. In *SCS Western Multiconference: Object-Oriented Simulation.* Society for Computer Simulation, San Diego, CA, 1992.

[Jac90] Jacob, G. K. What price, parallel Lisp? In *Proceedings of the EUROPAL Workshop on High Performance and Parallel Computing in Lisp*, November 1990.

[Jef85a] Jefferson, D. Implementation of Time Warp on the Caltech Hypercube". In *Proceedings of the SCS Distributed Simulation Conference*, pages 70–75, San Diego, January 1985. Society for Computer Simulation.

[Jef85b] Jefferson, D. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Jef87] Jefferson, D., et. al. Time Warp Operating System. In *Proc. Eleventh ACM Symposium on Operating System Principles*, pages 72–93, Austin, TX, 1987. ACM.

[Jen90] Jensen, K. Coloured Petri Nets: A High Level Language for System Design and Analysis. *Lecture Notes in Computer Science: Advances in Petri Nets 1990*, (483):342–416, 1990.

[Kar93] Karthik, S. and Abraham, J. A Framework for Distributed VLSI simulation on a network of Workstations. *Simulation*, 60(2):95–112, Feb 1993.

[Koc90] Koch B. et. al. Cache Coherency and Storage Management in a Persistent Object System. In *Proceedings of the Fourth Annual Conference on Persistent Object Systems*, pages 103–114. Morgan Kaufmann, 1990.

[Laf90] Lafore, R. *The Waite Group's C Programming Using Turbo C++*. MacMillan Computer Publishing, 1990.

[Lar93] Laret, L. Modelling Base and Model Formulation fo Knowledge System. In *Proceedings of the 1993 European Simulation Multiconference*, pages 227–231. Society for Computer Simulation International, University of Ghent, Belgium, 1993.

[Lee91] Lee, K. An Adaptive Extended Fuzzy Petri Net for Modeling of Fuzzy Production Systems. In *Proceedings of the 22nd Annual Pittsburgh Conference on Modeling and Simulation*, pages 2327–2334. The Modeling and Simulation Conference, University of Pittsburgh, 1991.

[Lina] Lindemann, C. Private Communication.

[Linb] Lindemann, C. and Shoen, F. A Deterministic and Stochastic Petri Net Model of a Consistency Protocol for a Distributed Shared Memory System. GMD Berlin. Unpublished.

[Lis79] Lister, A. M. *Fundamentals of Operating Systems.* MacMillian Press Ltd., 1979.

[Liv90] Livesy, M. Distributed Varimistic Concurrency Control in a Persistent Object Store. In *Fourth International Workshop on Persistent Object Systems*, pages 293–304. Morgan Kaufmann, 1990.

[Mar86] Marsan, A. J. et al. *Performance Models of Multiprocessor Systems.* MIT Press, 1986.

[Mar87] Marsan, M. and Chiola, G. On Petri Nets with Deterministic and Exponentially Distributed Firing Times. *Lecture Notes in Computer Science: Advances in Petri Nets 1987*, (266):132–145, 1987.

[Mar89] Marsan, M. Stochastic Petri Nets: An Elementary Introduction. In *Lecture Notes in Computer Science: Advances in Petri Nets 1989*, number 424, pages 1–29. Springer-Verlag, 1989.

[McA82] McArthur, D. and Klahr, P. The ROSS language manual. Note N-1854-AF, The RAND Corporation, September 1982.

[Mer92] Merrall, S. Plurals - A SIMD extension to Eulisp. Technical Report 92-59, School of Mathematics, University of Bath, 1992.

[Min86] Minsky, M. *The Society of the Mind.* Simon and Schuster, 1986.

[Mis86] Misra, J. Distributed Discrete-Event Simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[Mor90] Morrison, R. and Atkinson, M. P. Persistent Languages and Architectures. In *International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 9–28. Springer-Verlag, 1990.

[Mur89] Murata, T. Petri Nets: Properties, Analysis, and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Nor92] Norrie, D. and Kwok, A. Intelligent Agent Simulation of an Automated Guided Vehicle System. In *SCS Multiconference on Object-Oriented Simulation*, pages 94–100. Society for Computer Simulation, San Diego, CA, 1992.

[O'K86] O'Keefe, R. Simulation and Expert Systems—A Taxonomy and Some Examples. *Simulation*, 46(1):10–16, January 1986.

[Pad91] Padget, J. and Nuyens, G. (Eds.). The EuLisp Definition. School of Mathematics, University of Bath, Unpublished. 1991.

[Pet81] Peterson, J. L. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1981.

[Pit88] Pitt, L., and Valiant, L. G. Computational limitations on learning from examples. *ACM Journal*, 35:965–984, 1988.

[Rei90] Reiher, P, and Jefferson, D. Virtual Time Based Dynamic Load Management In The Time Warp Operating System. In *SCS Western Multiconference Distributed Simulation Track.* Society for Computer Simulation, San Diego, CA, 1990.

[Row86] Rowe, L. A. A Shared Object Hierarchy. In *International Workshop on Object-Oriented Database Systems*, pages 160–170. IEEE, 1986.

[Rum86] Rummelhart, D. E. and McClelland, J. L. *Parallel Distributed Processing - Volume 1: Foundations.* MIT Press, 1986.

[Rus79] Russell, E. C. Simulating with Processes and Resources in Simscript II.5. User's manual, CACI, 1979.

[Sab88] Sabot, G. *The Paralation Model.* MIT Press, 1988.

[Sav91] R. Savit. Chaos on the Trading Floor. *The New Scientist Guide to Chaos*, pages 174–184, 1991.

[Sou93] Soubra, S., et. al. Intelligent Simulation Environments: A First Application to Building Construction. In *Proceedings of the 1993 European Simulation Multiconference*, pages 222–226. Society for Computer Simulation International, University of Ghent, Belgium, 1993.

[Ste] Stern, H. Working paper comparing neural network to statistical techniques.

[Ste86] Sterling, L. and Shapiro, E. *The Art of Prolog.* MIT Press, 1986.

[Str91] Stroustrup, B. *The C++ Programming Language, Second Edition.* 1991.

[Thi88] Thinking Machines Corp. **Lisp Reference Manual.* 1988.

[Tou86] Touretzky, D. and Hinton, G. A distributed connectionist production system. Technical Report CMU-CSS-86-172, Carnegie-Mellon University, December 1986.

[Ull80] Ullman, J. *Principles of Database Systems.* Pitman, 1980.

[Ung93] Unger, B. Plenary Speech, European Simulation Multiconference, Lyon, France. 1993.

[Wan90] Wang, D. and Hsu, C. SLONN: A Simulation Language for modeling of Neural Networks. *Simulation*, pages 69–83, August 1990.

[Was89] Wasserman, P. *Neural Computing Theory and Practice.* Van Norstrand Rheinhold, 1989.

[Wei91] Weitzenfeld, A. Neural simulation language version 2.1. Technical Report 91-05, Center for Neural Engineering, University of Southern California, August 1991.

[Whi92a] Whitehurst, R. Simulation Utilizing an Integrated Object-oriented and Rule-based Approach. In *SCS Western Multiconference: Object-Oriented Simulation*, pages 88–93. Society for Computer Simulation, San Diego, CA, 1992.

[Whi92b] Whitehurst, R. A. Simulation Utilizing an Integrated Object-Oriented and Rule-based Approach. In *SCS Western Multiconference: Object-Oriented Simulation*, pages 75–79. Society for Computer Simulation, San Diego, CA, 1992.

[Wil87] Williams, I., Wolczko, M., and Hopkins, T. Dynamic Grouping in an Object-Oriented Virtual Memory Hierarchy. In *Proceedings of The European Conference on Object-Oriented Programming*, pages 79–85, 1987.

[Wil90] Williams, I. and Wolczko, M. An Object-Based Memory Architecture. In *Proceedings of the Fourth Annual Conference on Persistent Object Systems*, pages 114–130. Morgan Kaufmann, 1990.